UNIVERSITÀ DEGLI STUDI DI PALERMO

dipartimento di ingegneria unipa

Object Oriented Programming in Javascript

Corso di Programmazione Web e Mobile

a.a. 2021/2022

Prof. Roberto Pirrone

LABORATORIO DI INTERAZIONE UOMO-MACCHINA
CHILAB

Sommario

- Dichiarazione degli oggetti
  - Polimorfismo
  - `this`
- Mappe
- Simboli
- Iteratori
- `get` e `set` e `static`
- Ereditarietà

Dichiarazione degli oggetti

- Javascript dichiara gli oggetti aggiungendo proprietà ad un oggetto inizialmente vuoto
- Non esiste un vero incapsulamento: non ci sono proprietà private
- Tali oggetti sono direttamente ereditati da `Object`

```
var person = {};

// person è una istanza di Object
person.name = 'Roberto';
person.surname = 'Pirrone';
person.age = 25;
person.isALiar = function() { return true; }
```

```
> person
{
  name: 'Roberto',
  surname: 'Pirrone',
  age: 25,
  isALiar: [Function (anonymous)]
}
```

Dichiarazione degli oggetti

- La dichiarazione di una «classe» risiede nella definizione della proprietà `prototype` della classe stessa
- All'interno di `prototype` si conservano le proprietà che sono invariate per tutte le istanze della classe
- Le proprietà che cambiano da istanza a istanza sono definite direttamente all'interno di ogni oggetto

Dichiarazione degli oggetti

- I prototipi si possono creare in vari modi
  - Dichiarazione esplicita della funzione costruttore
  - Metodo `Object.create()`
  - Costrutto `class`

Dichiarazione degli oggetti

```
function Persons(name, ...args) {

        this.name = name; // this --> riferimento all'oggetto che fa
                          // da contesto all'esecuzione

        this.surname = (args && typeof args[0] == 'string' && args[0]) || null;
        this.age = (args && typeof args[0] == 'number' && args[0]) ||
                   (args && typeof args[1] == 'number' && args[1]) || null;
}

Persons.prototype.isALiar = function() { return false; }

let jack = new Persons('Jack', 'London', 87);
```

Dichiarazione degli oggetti

- Polimorfismo
  - Un *codice polimorfo* è tale per cui, stabilita una certa interfaccia per i dati su cui opera, lavora in maniera trasparente con tutto ciò che espone quell'interfaccia
  - Il codice polimorfo possiamo ottenerlo con liste di argomenti variabili per adattarsi a diverse tipi di argomenti
  - Analogamente potremo condizionare le istruzioni di `return` per gestire diversi tipi del risultato

Dichiarazione degli oggetti

- Polimorfismo
  - Esempio: il costruttore `String()` trasforma qualunque oggetto in stringa eseguendo il suo metodo `toString()`
  - È bene ridefinire `toString()` per comunicare informazioni pertinenti sugli oggetti che creiamo.

Dichiarazione degli oggetti

- `this`
  - È il riferimento all'oggetto di contesto per un metodo, proprietà o altro oggetto
  - Le funzioni possono impostare esplicitamente il loro contesto tramite il metodo `bind()` della classe `Function`

Dichiarazione degli oggetti

- `this` in `node` è un opportuno oggetto che definisce l'ambiente di elaborazione e contiene le nostre dichiarazioni

- `this` nel browser è la `Window` corrispondente alla finestra corrente

```
> var a = true
undefined
> function pippo(){return 6;}
undefined
> this
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]:
[Function (anonymous)]
  },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]:
[Function (anonymous)]
  },
  a: true,
  pippo: [Function: pippo]
}
```

Dichiarazione degli oggetti

```
> jack
Persons { name: 'Jack', surname: 'London', age: 87 }
> jack.
jack.__defineGetter__          jack.__defineSetter__          jack.__lookupGetter__
jack.__lookupSetter__
jack.__proto__                 jack.hasOwnProperty            jack.isPrototypeOf
jack.propertyIsEnumerable
jack.toLocaleString            jack.toString                  jack.valueOf

jack.constructor               jack.isALiar

jack.age                       jack.name                      jack.surname

> jack.constructor
[Function: Persons]
> jack instanceof Persons
false
> jack instanceof Object
true
```

Il metodo sta nel prototipo e non tra le proprietà

Dichiarazione degli oggetti

```
// Uso di Object.create()
const protoPeople = {
name: 'Fred',
surname: 'Flinstone',
salutation: function() {
console.log(`Hi everybody! My name is ${this.name} ${this.surname}`);
}
}

// il nuovo oggetto sovrascrive le proprietà del prototipo
let barney = Object.create(protoPeople);
barney.name = 'Barney';
barney.surname = 'Rumble';
```

Dichiarazione degli oggetti

```
> barney
{ name: 'Barney', surname: 'Rumble' }
> barney.
barney.__defineGetter__      barney.__defineSetter__      barney.__lookupGetter__
barney.__lookupSetter__      barney.__proto__             barney.constructor
barney.hasOwnProperty        barney.isPrototypeOf         barney.propertyIsEnumerable
barney.toLocaleString        barney.toString              barney.valueOf

barney.salutation

barney.name                  barney.surname

> barney.constructor
[Function: Object]
> barney instanceof Object
true
```

Dichiarazione degli oggetti

```
// Persona è una classe secondo la definizione
// ECMAScript 2015 e successive
class Persona {

        constructor(name, surname) {
                this.name = name;
                this.surname = surname;
        }

        sayHello = function() {
                console.log(`Hi, my name is ${this.name} ${this.surname}`)
        }
}

let fred = new Persona('Fred', 'Flinstone');
```

Dichiarazione degli oggetti

```
> fred
Persona {
  sayHello: [Function: sayHello],
  name: 'Fred',
  surname: 'Flinstone'
}
> fred.
fred.__defineGetter__        fred.__defineSetter__
fred.__lookupGetter__        fred.__lookupSetter__
fred.__proto__               fred.hasOwnProperty
fred.isPrototypeOf           fred.propertyIsEnumerable
fred.toLocaleString          fred.toString
fred.valueOf

fred.constructor

fred.name                    fred.sayHello              fred.surname

> fred instanceof Persona
true
> fred instanceof Object
true
> Persona.prototype.isPrototypeOf(fred)
true
> Persons.prototype.isPrototypeOf(jack)
false
```

`fred` è un vero e proprio oggetto `Persona`

Mappe

- L'utilizzo più frequente di un oggetto è come sequenza di coppie chiave-valore
- `Object.keys()` e `Object.values()` forniscono l'accesso ai due array delle chiavi e dei valori di qualunque oggetto

```
> Object.values(fred)
[ [Function: sayHello], 'Fred', 'Flinstone' ]
> Object.keys(fred)
[ 'sayHello', 'name', 'surname' ]
> fred.hasOwnProperty('toString')
false
> fred.hasOwnProperty('name')
true
```

Ritorna `true` se la proprietà si trova nell'oggetto e non nel prototipo

Mappe

- La classe `Map` è studiata apposta per gestire direttamente le coppie chiave-valore senza che queste siano in realtà oggetti con prototipi e/o costruttori

- Una mappa è un iterabile fatto di array da due elementi contenenti la chiave ed il rispettivo valore

- Essa espone l'interfaccia i cui metodi principali sono `set()`, `get()` e `has()`

Mappe

```
let archive = new Map();

archive.set('desktop', 56);
archive.set('laptop', 23);
archive.set('smartphone', 12);
archive.set('headset', 24);
```

```
> archive
Map(4) {
   'desktop' => 56,
   'laptop' => 23,
   'smartphone' => 12,
   'headset' => 24
}
> for (let x of archive)
... console.log(x)
[ 'desktop', 56 ]
[ 'laptop', 23 ]
[ 'smartphone', 12 ]
[ 'headset', 24 ]
undefined
> for (let x in archive)
... console.log(x)
undefined
> archive.has('laptop')
true
> archive.get('smartphone')
12
```

Simboli

- I nomi delle proprietà degli oggetti possono essere stringhe o «simboli»
- I simboli si definiscono attraverso la funzione `Symbol()` e sono univoci all'interno di un programma
- `Symbol()` si comporta com un costruttore, ma non vuole la parola chiave `new`

```
// Due simboli con la stessa definizione
let Sym1 = Symbol("Sym")
let Sym2 = Symbol("Sym")

console.log(Sym1 === Sym2)
// restituisce false
```

Simboli

- Una proprietà definita come simbolo è accessibile con la notazione `[…]`, *ma non è enumerabile tra le proprietà dell'oggetto*
  - Non viene elencata con il costrutto `for ... In`
  - Si ottiene attraverso `Object.getOwnPropertySybols()`
- Javascript mantiene un registro dei simboli attraverso le chiavi stringa che vengono usate per crearli; la gestione avviene attraverso i metodi

  `Symbol.keyFor(`*`simbolo`*`) Symbol.for(`*`chiaveStringa`*`)`
- `Symbol.toString()` ovvero `Symbol.description` forniscono una descrizione estesa del simbolo

Iteratori

- Un iteratore si definisce attraverso il simbolo speciale `Symbol.iterator`

```
var nomeIteratore =
        oggettoIterabile[Symbol.iterator]();
```

- L'iteratore ha una interfaccia definita dal metodo `next()` che ritorna un oggetto:

```
{value: valoresuccessivo, done: true/false}
```

Iteratori

```
let people = [new Persona('Fred', 'Flinstone'),
new Persona('Barney', 'Rumble'),
new Persona('Wilma', 'Flinstone')
];

let peopleIterator = people[Symbol.iterator]();
```

```
> peopleIterator.next()
{
  value: Persona {
    sayHello: [Function: sayHello],
    name: 'Fred',
    surname: 'Flinstone'
  },
  done: false
}
> peopleIterator.next()
{
  value: Persona {
    sayHello: [Function: sayHello],
    name: 'Barney',
    surname: 'Rumble'
  },
  done: false
}
> peopleIterator.next()
{
  value: Persona {
    sayHello: [Function: sayHello],
    name: 'Wilma',
    surname: 'Flinstone'
  },
  done: false
}
> peopleIterator.next()
{ value: undefined, done: true }
```

Iteratori

```
class myPeopleIterator {
        constructor(peopleArray) {
        this.count = 0;
        this.peopleArray = peopleArray;
        }

        next() {
                if (this.count == this.peopleArray.length)
                        return { value: undefined, done: true };
                else {
                        let obj = {
                                value: `Hi! My name is \
                                ${this.peopleArray[this.count].name} \
                                ${this.peopleArray[this.count].surname}!`,
                                done: false
                                };
                        this.count++;
                        return obj;
                }

        }
}

people[Symbol.iterator] = function() { return new myPeopleIterator(this) }
```

```
> for (x of people)
... console.log(x)
Hi! My name is Fred Flinstone!
Hi! My name is Barney Rumble!
Hi! My name is Wilma Flinstone!
```

Iteratori

- Per una classe l'iteratore custom si definisce *all'interno del prototipo della classe*

```
MyClass.prototype[Symbol.iterator] = function() {

                                    return new myPeopleIterator(this)
                                    }
```

`get` `set` e `static`

- `get` e `set` creano dei metodi i cui nomi sono utilizzati direttamente come proprietà in operazioni di assegnamento

- Lo scopo è quello di creare una interfaccia di accesso all'oggetto che non utilizzi direttamente la rappresentazione interna dei dati che però non possono essere resi privati

- È possibile rendere «riservate» le rappresentazioni interne di singoli oggetti attraverso il metodo `Object.defineProperty()`

get set e static

```
> Temperature.fromFahrenheit(34)
Temperature { celsius:
1.1111111111111112 }
> temp.celsius
25
> temp.celsius=34
34
> temp.celsius
25
> Object.keys(temp)
[]
> temp.fahrenheit
77
> temp.fahrenheit=89
Uncaught:
TypeError: …
```

```
class Temperature {
        constructor(celsius) {
                this.celsius = celsius;
        }

        get fahrenheit() {
                return this.celsius * 1.8 + 32;
        }
        set fahrenheit(value) {
                this.celsius = (value - 32) / 1.8;
        }
        static fromFahrenheit(value) {
                return new Temperature((value - 32) / 1.8);
        }
}

let temp = new Temperature(25);
Object.defineProperty(temp, 'celsius', {
        writable: false,
        enumerable: false
});
```

Ereditarietà

- Il meccanismo dell'ereditarietà si basa sulla direttiva `extends` che stabilisce che la nuova classe non ha il prototipo di Object, ma quello della classe da cui eredita

```
class classeDerivata extends classeBase {
        …
}
```

Ereditarietà

- È necessario accedere al contesto della superclasse per invocare il suo costruttore ed i suoi metodi all'interno della classe derivata. Il contesto della superclasse viene fornito da `super`

```
class classeDerivata extends classeBase {
        constructor(x, y){
            super(x,y);
            … }
        …
}
```

Ereditarietà

- Il contesto `this` della classe derivata sarà utile per proprietà e metodi propri di quest'ultima e non della superclasse

```
class classeDerivata extends classeBase {
        constructor(x, y, a){
            this.alpha = a;
            super(x,y);
            … }
        …
}
```

Ereditarietà

- Sarà ovviamente possibile ridefinire i metodi della superclasse nella classe derivata

```
class classeDerivata extends classeBase {
        constructor(x, y, a){
            … }
        myMethod(x,y){
            if(this.alpha > 3)
                super.myMethod(x,y);
            else … ;
        … }
        …
}
```