La programmazione OOP in JavaScript

La programmazione orientata agli oggetti (OOP) in JavaScript è un paradigma di programmazione che si basa sull'idea di modellare i dati e il comportamento tramite **oggetti**. Sebbene JavaScript sia nato come un linguaggio orientato ai prototipi, supporta appieno la programmazione OOP moderna con l'introduzione di **classi** in ECMAScript 2015 (ES6).

Sommario

1. Concetti Fondamentali di OOP

- **Oggetti**: Rappresentano entità con proprietà (dati) e metodi (comportamenti).
- **Classi**: Modellano un tipo di oggetto, definendo proprietà e metodi condivisi.
- **Ereditarietà**: Consente a una classe di ereditare proprietà e metodi da un'altra classe.
- **Incapsulamento**: Consente di nascondere dettagli interni e esporre un'interfaccia pubblica.
- **Polimorfismo**: Consente di usare lo stesso metodo in contesti diversi.

2. Creazione di Oggetti

Con oggetti letterali:

```
const persona = {
    nome: 'Mario',
    saluta: function() {
        console.log(`Ciao, mi chiamo ${this.nome}`);
    }
};

persona.saluta(); // Output: Ciao, mi chiamo Mario
```

Con funzioni costruttrici:

Prima di ES6, questa era una tecnica comune per creare oggetti:

```
function Persona(nome) {
    this.nome = nome;
    this.saluta = function() {
        console.log(`Ciao, mi chiamo ${this.nome}`);
    };
}

const mario = new Persona('Mario');
mario.saluta(); // Output: Ciao, mi chiamo Mario
```

3. Classi (Introduzione in ES6)

Le **classi** forniscono una sintassi più chiara e moderna per lavorare con oggetti e costruttori.

Definizione di una classe:

```
class Persona {
    constructor(nome) {
        this.nome = nome; // proprietà
    }

    saluta() { // metodo
        console.log(`Ciao, mi chiamo ${this.nome}`);
    }
}

const mario = new Persona('Mario');
mario.saluta(); // Output: Ciao, mi chiamo Mario
```

4. Ereditarietà

La parola chiave `extends` consente di creare classi derivate da altre classi.

```
class Animale {
    constructor(nome) {
        this.nome = nome;
    }

    verso() {
        console.log(`${this.nome} fa un verso.`);
    }
}

class Cane extends Animale {
    verso() {
        console.log(`${this.nome} abbaia.`);
    }
}

const fido = new Cane('Fido');
fido.verso(); // Output: Fido abbaia.
```

5. Incapsulamento

JavaScript supporta l'incapsulamento tramite l'uso di convenzioni e parole chiave come `#` per proprietà private (introdotte in ES2022).

Proprietà pubbliche:

```
class Persona {
    constructor(nome) {
        this.nome = nome; // Proprietà pubblica
    }
}
```

Proprietà private:

```
class ContoBancario {
    #saldo; // Proprietà privata

    constructor(saldoIniziale) {
        this.#saldo = saldoIniziale;
    }

    deposita(importo) {
        this.#saldo += importo;
        console.log(`Nuovo saldo: ${this.#saldo}`);
    }
}

const conto = new ContoBancario(1000);
conto.deposita(500); // Output: Nuovo saldo: 1500
// console.log(conto.#saldo); // Errore: Proprietà privata
```

6. Polimorfismo

Il polimorfismo permette alle sottoclassi di ridefinire metodi delle classi base.

```
class Forma {
    disegna() {
        console.log('Disegno una forma generica.');
    }
}

class Cerchio extends Forma {
    disegna() {
        console.log('Disegno un cerchio.');
    }
}

class Rettangolo extends Forma {
    disegna() {
        console.log('Disegno un rettangolo.');
    }
}

const forme = [new Cerchio(), new Rettangolo(), new Forma()];
forme.forEach(forma => forma.disegna());
// Output:
// Disegno un cerchio.
// Disegno un rettangolo.
// Disegno una forma generica.
```

7. Prototipi e OOP Basato sui Prototipi

Anche con le classi, JavaScript rimane un linguaggio basato sui **prototipi**. Ogni oggetto ha una proprietà interna `[[Prototype]]`, accessibile con `Object.getPrototypeOf` o tramite la proprietà `__proto__`.

Esempio:

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

Persona.prototype.saluta = function() {
    console.log(`Ciao, mi chiamo ${this.nome}`);
};

const mario = new Persona('Mario');
mario.saluta(); // Output: Ciao, mi chiamo Mario
```

8. Paragone tra Funzioni Costruttrici e Classi

Con funzione costruttrice:

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

Persona.prototype.saluta = function() {
    console.log(`Ciao, mi chiamo ${this.nome}`);
};
```

Con classe:

```
class Persona {
    constructor(nome) {
        this.nome = nome;
    }

    saluta() {
        console.log(`Ciao, mi chiamo ${this.nome}`);
    }
}
```

9. Conclusione

JavaScript consente di implementare l'OOP in modo flessibile, utilizzando:

- **Oggetti letterali** per casi semplici.
- **Prototipi** per personalizzazioni avanzate.
- **Classi** per una sintassi moderna e leggibile.

Esempio di programmazione orientata agli oggetti (OOP) in JavaScript

Esempio di programmazione orientata agli oggetti (OOP) in JavaScript, utilizzando un **oggetto letterale** per modellare una persona con proprietà e metodi.

1. L'oggetto `persona`

L'oggetto `persona` incarna i principi base dell'OOP:

- **Proprietà**: Sono i dati che caratterizzano l'oggetto.
  - `nome`: rappresenta il nome della persona.
  - `eta`: rappresenta l'età della persona.
- **Metodi**: Sono le funzioni che definiscono il comportamento dell'oggetto.
  - `saluta`: è un metodo che stampa un messaggio di saluto utilizzando la proprietà `nome`.

Struttura dell'oggetto:

```
let persona = {
    nome: "Luca", // Proprietà: descrive il nome
    eta: 30,      // Proprietà: descrive l'età
    saluta: function() { // Metodo: descrive un comportamento
        console.log("Ciao, sono " + this.nome);
    }
};
```

2. OOP nel codice

Proprietà

Le proprietà `nome` e `eta` rappresentano **gli attributi** o **lo stato** dell'oggetto. In un modello OOP, queste proprietà descrivono le caratteristiche uniche di una persona.

Metodo

Il metodo `saluta` rappresenta il **comportamento** dell'oggetto. Utilizza la parola chiave `this` per accedere alle proprietà dell'oggetto stesso:

- `this.nome` si riferisce alla proprietà `nome` dell'oggetto `persona`.

Esempio di chiamata:

```
persona.saluta(); // Output: Ciao, sono Luca
```

In questo caso, `this.nome` viene risolto come `"Luca"`, poiché si riferisce alla proprietà `nome` dell'oggetto `persona`.

3. Trasformazione in una Classe

Per rendere il codice più conforme ai principi OOP moderni, possiamo rappresentare lo stesso concetto utilizzando una classe.

Codice:

```
class Persona {
    constructor(nome, eta) {
        this.nome = nome; // Inizializza la proprietà nome
        this.eta = eta;   // Inizializza la proprietà eta
    }

    saluta() { // Metodo per salutare
        console.log("Ciao, sono " + this.nome);
    }
}

// Creazione di un'istanza
let persona = new Persona("Luca", 30);
persona.saluta(); // Output: Ciao, sono Luca
```

In questa versione:

1. La classe `Persona` è una struttura astratta che definisce le proprietà (`nome`, `eta`) e il metodo (`saluta`) che tutte le istanze avranno.
2. `this` si riferisce sempre all'istanza attuale dell'oggetto, proprio come nell'oggetto letterale.

4. Differenze con l'oggetto letterale

Oggetto Letterale	Classe
Non crea un "modello" generale; l'oggetto è definito direttamente.	Fornisce un modello per creare più oggetti simili.
Adatto per oggetti unici e semplici.	Ideale per strutture riutilizzabili e complesse.
Nessun costruttore.	Utilizza il metodo `constructor` per inizializzare le proprietà.
Non supporta l'ereditarietà nativa.	Supporta ereditarietà con `extends`.

---

5. Utilizzo dell'Oggetto Letterale

L'approccio con un **oggetto letterale** è utile quando:

- Non hai bisogno di creare più oggetti dello stesso tipo.
- Il tuo oggetto è relativamente semplice e unico. Esempio:

```
let macchina = {
    marca: "Fiat",
    modello: "Panda",
    accendi: function() {
        console.log("La macchina è accesa!");
    }
};
```

Concetti fondamentali della programmazione orientata agli oggetti (OOP) in JavaScript

Concetti fondamentali della **programmazione orientata agli oggetti (OOP)** in JavaScript, con esempi pratici per chiarire ogni elemento.

1. Oggetti

Gli oggetti rappresentano **entità** che combinano dati (**proprietà**) e comportamenti (**metodi**). In JavaScript, gli oggetti possono essere creati in diversi modi, ma l'idea fondamentale è che un oggetto rappresenta un'istanza concreta di un concetto.

Creazione di un oggetto:

```
let persona = {
    nome: "Mario",             // Proprietà
    eta: 30,                   // Proprietà
    saluta: function() {       // Metodo
        console.log("Ciao, sono " + this.nome);
    }
};

persona.saluta(); // Output: Ciao, sono Mario
```

- **Proprietà**: `nome` e `eta` rappresentano i dati.
- **Metodo**: `saluta` definisce il comportamento.

Gli oggetti sono la base di tutto in JavaScript. Ogni array, funzione o anche un oggetto globale come `Math` è un oggetto.

2. Classi

Le classi in JavaScript sono una struttura che serve a **modellare oggetti**. Esse definiscono un tipo di oggetto con proprietà e metodi condivisi, che possono essere istanziati tramite la parola chiave `new`.

Creazione di una classe:

```
class Persona {
    constructor(nome, eta) { // Costruttore: inizializza le proprietà
        this.nome = nome;
        this.eta = eta;
    }

    saluta() { // Metodo: comportamento comune
        console.log("Ciao, sono " + this.nome);
    }
}

// Creazione di un'istanza (oggetto) dalla classe
let mario = new Persona("Mario", 30);
let luca = new Persona("Luca", 25);

mario.saluta(); // Output: Ciao, sono Mario
```

```
luca.saluta();  // Output: Ciao, sono Luca
```

Le **classi** sono utili quando si vogliono creare più oggetti simili con comportamenti e dati comuni.

3. Ereditarietà

L'ereditarietà consente a una classe di derivare da un'altra classe, **ereditandone** proprietà e metodi. Questo permette di creare gerarchie di classi e riutilizzare codice.

Creazione di una classe derivata:

```
class Animale {
    constructor(nome) {
        this.nome = nome;
    }

    verso() {
        console.log(`${this.nome} fa un verso.`);
    }
}

class Cane extends Animale { // 'Cane' eredita da 'Animale'
    verso() { // Override del metodo
        console.log(`${this.nome} abbaia.`);
    }
}

let animale = new Animale("Animale generico");
animale.verso(); // Output: Animale generico fa un verso.

let fido = new Cane("Fido");
fido.verso(); // Output: Fido abbaia.
```

- La classe `Cane` eredita da `Animale`, ma può anche **sovrascrivere** i metodi (`verso`) o aggiungerne di nuovi.

4. Incapsulamento

L'incapsulamento consiste nel **nascondere i dettagli interni** di un oggetto e nell'esporre solo un'interfaccia pubblica per interagirvi. Questo protegge i dati dell'oggetto da modifiche accidentali.

Proprietà pubbliche e private:

In JavaScript, le proprietà private sono identificate con il prefisso `#`, introdotto in ES2022.

```
class ContoBancario {
    #saldo; // Proprietà privata

    constructor(saldoIniziale) {
        this.#saldo = saldoIniziale;
    }

    deposita(importo) {
        if (importo > 0) {
            this.#saldo += importo;
            console.log(`Deposito effettuato. Nuovo saldo: ${this.#saldo}`);
        } else {
```

```
            console.log("Importo non valido.");
        }
    }

    // Metodo pubblico per leggere il saldo
    getSaldo() {
        return this.#saldo;
    }
}

let conto = new ContoBancario(1000);
conto.deposita(500); // Output: Deposito effettuato. Nuovo saldo: 1500
console.log(conto.getSaldo()); // Output: 1500
// console.log(conto.#saldo); // Errore: Proprietà privata
```

L'uso delle proprietà private (`#saldo`) garantisce che i dati non siano accessibili o modificabili direttamente, ma solo attraverso metodi controllati.

5. Polimorfismo

Il polimorfismo permette di utilizzare lo stesso metodo su oggetti di classi diverse, ottenendo comportamenti differenti. È strettamente legato all'ereditarietà.

Esempio di polimorfismo:

```
class Forma {
    disegna() {
        console.log("Disegno una forma generica.");
    }
}

class Cerchio extends Forma {
    disegna() {
        console.log("Disegno un cerchio.");
    }
}

class Rettangolo extends Forma {
    disegna() {
        console.log("Disegno un rettangolo.");
    }
}

// Funzione che utilizza il polimorfismo
function disegnaForma(forma) {
    forma.disegna();
}

let forme = [new Cerchio(), new Rettangolo(), new Forma()];
forme.forEach(forma => disegnaForma(forma));
// Output:
// Disegno un cerchio.
// Disegno un rettangolo.
// Disegno una forma generica.
```

In questo esempio, il metodo `disegna` ha implementazioni diverse a seconda della classe specifica, ma viene chiamato allo stesso modo.

Tabella Riassuntiva

Concetto	Descrizione	Esempio
Oggetti	Rappresentano entità con dati (proprietà) e comportamenti (metodi).	`{ nome: "Mario", saluta: function() { console.log(this.nome); } }`
Classi	Modelli per creare oggetti con dati e comportamenti condivisi.	`class Persona { constructor(nome) { this.nome = nome; } }`
Ereditarietà	Una classe può derivare da un'altra, ereditandone proprietà e metodi.	`class Cane extends Animale { ... }`
Incapsulamento	Nasconde dettagli interni ed espone solo un'interfaccia pubblica.	`#saldo` come proprietà privata; accessibile solo tramite metodi come `getSaldo()`.
Polimorfismo	Consente di usare lo stesso metodo su oggetti di classi diverse.	`forma.disegna()` chiama metodi diversi in base alla classe dell'oggetto (`Cerchio`, `Rettangolo`).