Oggetti e classi

Niccolò Maltoni
niccolo.maltoni@diennea.com

Astrazione

In logica

> Metodo per ottenere concetti universali da oggetti particolari, mettendo da parte le loro caratteristiche specifiche.

In informatica

> Applicazione del metodo logico di astrazione nella strutturazione della descrizione dei sistemi informatici complessi, per facilitarne la progettazione e manutenzione o la stessa comprensione.

Ogni linguaggio di programmazione introduce un livello di astrazione per rappresentare il problema reale.

Paradigmi di programmazione

- C, Pascal: **Programmazione imperativa e procedurale**

  > Computing function/procedure over data structures

- Lisp, Haskell: **Programmazione funzionale**

  > Everything is a function

- Java, C++, C#: **Programmazione orientata agli oggetti**

  > Everything is an object (OO Programming)

JavaScript supporta tutti e tre i paradigmi, ma oggi ci concentriamo sull'**OOP**.

Oggetti

Nella lezione 2 abbiamo visto un tipo di dato fondamentale: **l'oggetto**.

Abbiamo visto l'oggetto come una struttura dati con proprietà (chiave-valore) e metodi (funzioni).

```
const persona = {
  nome: "Mario",
  eta: 30,

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

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

Problema: se vogliamo creare 10 persone diverse, dobbiamo riscrivere tutto 10 volte!

```
const persona1 = { nome: "Mario", eta: 30, saluta() {...} };
const persona2 = { nome: "Luigi", eta: 28, saluta() {...} };
const persona3 = { nome: "Anna", eta: 25, saluta() {...} };
// ...ripetitivo e soggetto a errori!
```

Il pattern costruttore

La soluzione: creare una **funzione costruttore** che produce oggetti simili:

```
function Persona(nome, eta) {
  this.nome = nome;
  this.eta = eta;

  this.saluta = function() {
    console.log("Ciao, sono " + this.nome);
  };
}

// Creare istanze con "new"
const mario = new Persona("Mario", 30);
const luigi = new Persona("Luigi", 28);

mario.saluta();  // "Ciao, sono Mario"
luigi.saluta();  // "Ciao, sono Luigi"
```

Per **convenzione**, i costruttori iniziano con la **maiuscola** (`Persona`, non `persona`).

Operatore `new`: cosa fa esattamente?

Quando scriviamo `new Persona("Mario", 30)`, JavaScript:

1. **Crea un oggetto vuoto** `{}`
2. **Imposta** `this` nel costruttore = nuovo oggetto
3. **Esegue il costruttore** (aggiunge proprietà a `this`)
4. **Ritorna** `this` automaticamente

```
// Quello che scriviamo
const mario = new Persona("Mario", 30);

// Quello che JavaScript fa internamente
function Persona(nome, eta) {
  // this = {};  (implicito)

  this.nome = nome;
  this.eta = eta;
  this.saluta = function() {
    console.log("Ciao, sono " + this.nome);
  };

  // return this;  (implicito)
}
```

this

`this` è una parola chiave speciale che si riferisce all'**oggetto corrente** (quello su cui stiamo operando).

Il valore di `this` viene valutato al momento dell'esecuzione. Ad esempio, la stessa funzione potrebbe avere diversi `this` quando viene chiamata da oggetti diversi:

```
const user = { name: "John" };
const admin = { name: "Admin" };

function sayHi() {
  alert(this.name);
}

// utilizziamo la stessa funzione su due oggetti
user.f = sayHi;
admin.f = sayHi;

// queste chiamate hanno un this diverso!
user.f();  // John  (this == user)
admin.f(); // Admin  (this == admin)
```

⚠ `this` non ha limiti

Diversamente da molti altri linguaggi, in JavaScript la parola chiave `this` può essere usata in qualsiasi funzione, anche se non si tratta di un metodo.

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined in modalità strict, l'oggetto globale window in modalità non strict
```

Esercizio: Costruttore Libro

Consiglio

Crea una funzione costruttore `Libro` con:

- Proprietà: `titolo`, `autore`, `pagine`
- Metodo: `descrivi()` che stampa “Titolo di Autore (X pagine)”

Crea 3 istanze diverse e chiamane i metodi.

Classi ES6: sintassi moderna

A partire da ES6 (2015), JavaScript introduce la sintassi **`class`**:

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

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

const mario = new Persona("Mario", 30);
mario.saluta();  // "Ciao, sono Mario"
```

Nonostante si tratti principalmente di **zucchero sintattico**, ci sono vantaggi rispetto al costruttore classico:

- Introdotte per mappare i concetti di OOP come **ereditarietà** e **incapsulamento**
- Sintassi più pulita e familiare (simile a Java, C#, Python)
- Metodi definiti una sola volta (anche in memoria! Non sono duplicati per ogni istanza)
- Funzionalità avanzate (ereditarietà, getter/setter, static)

`constructor`: il metodo speciale

Il metodo **`constructor`** viene chiamato automaticamente quando creiamo un'istanza con `new`:

```
class Persona {
  constructor(nome, eta) {
    console.log("Sto creando una persona!");
    this.nome = nome;
    this.eta = eta;
  }
}

const mario = new Persona("Mario", 30);
// Stampa: "Sto creando una persona!"
```

Regole:

- Può esserci **un solo** `constructor` per classe
- Se non lo scrivi, JavaScript ne crea uno vuoto automaticamente
- Serve per **inizializzare** le proprietà dell'istanza

Metodi di istanza

I metodi definiti nella classe sono **condivisi** da tutte le istanze:

```
class Contatore {
  constructor() {
    this.valore = 0;
  }

  incrementa() {
    this.valore++;
  }

  mostra() {
    console.log("Valore: " + this.valore);
  }
}

const c1 = new Contatore();
const c2 = new Contatore();

c1.incrementa();
c1.incrementa();

c1.mostra();  // "Valore: 2"
c2.mostra();  // "Valore: 0" (indipendente!)
```

Ogni istanza ha il **proprio stato** (`valore`), ma condivide i **metodi**.

Metodi statici

I metodi **statici** appartengono alla **classe**, non alle istanze:

```
class Matematica {
  static raddoppia(x) {
    return x * 2;
  }

  static somma(a, b) {
    return a + b;
  }
}

// Chiamata sulla classe (NON su istanze)
console.log(Matematica.raddoppia(5));    // 10
console.log(Matematica.somma(3, 7));     // 10

// const m = new Matematica();
// m.raddoppia(5);  // Errore! raddoppia non esiste sull'istanza
```

Quando usare `static`:

- Utility functions (es. `Math.random()`, `Array.isArray()`)
- Factory methods (metodi che creano istanze)
- Funzioni che non dipendono dallo stato dell'istanza

Esercizio: Classe Prodotto

Consiglio

Crea una classe `Prodotto` con:

- `constructor(nome, prezzo)`
- Metodo di istanza: `descrivi()`
- Metodo statico: `confrontaPrezzo(prod1, prod2)` che ritorna il più economico

Crea due prodotti e confrontali.

Incapsulamento

L’**incapsulamento** nella programmazione orientata agli oggetti consiste nel nascondere i dettagli interni di un oggetto e fornire un’interfaccia pubblica per accedere ai suoi metodi e attributi. Due ingredienti:

1. “**Impacchettamento**” dati + *funzioni per manipolarli*
2. Controllo d’accesso e **information hiding**

Ogni classe dovrebbe esporre solo quei (pochi) metodi necessari a interagire con le sue istanze in modo completo. Il resto dovrebbe essere mantenuto privato.

Per fare un’analogia con il mondo reale, un bancomat nasconde il meccanismo interno. L’utente usa i pulsanti (interfaccia pubblica), non accede direttamente ai soldi.

Information hiding

L'**information hiding** è ciò che sta alla base dell'incapsulamento.

Il client (chi usa l'oggetto) conosce **cosa può fare** (l'interfaccia), **non come funziona** (i dettagli interni).

```
function Conto(saldoIniziale) {
  this.saldo = saldoIniziale;

  // Metodo pubblico che dovrebbe gestire i controlli
  this.deposita = function(importo) {
    if (typeof importo !== "number" || importo <= 0) return;
    this.saldo += importo;
  };
}

const conto = new Conto(1000);

conto.deposita(200);

// nulla impedisce di bypassare l'API pubblica e modificare `saldo` direttamente
conto.saldo = -500;
```

La soluzione è rendere privati i campi e controllare con getter/setter

Campi privati con

In JavaScript moderno (ES2022), usiamo # per dichiarare proprietà private:

```
class ContoCorrente {
  #saldo;  // Campo privato

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

  deposita(importo) {
    if (importo > 0) {
      this.#saldo += importo;
    }
  }

  getSaldo() {
    return this.#saldo;  // Accesso controllato
  }
}

const conto = new ContoCorrente(1000);
conto.deposita(500);
console.log(conto.getSaldo());  // 1500

console.log(conto.#saldo);  // SyntaxError: Private field '#saldo' must be declared in an enclosing class
```

Vantaggi: impossibile modificare `#saldo` dall'esterno. Il controllo è centralizzato nei metodi.

Getter e setter

Con **getter** e **setter**, creiamo proprietà “virtuali” con logica di controllo:

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

  get eta() {
    return new Date().getFullYear() - this.annoNascita;
  }

  set eta(valore) {
    if (valore < 0 || valore > 150) return;
    this.annoNascita = new Date().getFullYear() - valore;
  }
}

const mario = new Persona("Mario", 1994);
console.log(mario.eta);       // 31 (calcolato)
mario.eta = 25;               // Modifica via setter
console.log(mario.annoNascita);  // 2001 (aggiornato)
```

Nota: dall’esterno sembrano proprietà normali, ma sono metodi con logica.

Esercizio: Classe Temperatura

Proviamo l'incapsulamnento

Crea una classe `Temperatura` con:

- Campo privato `#celsius`
- Getter `fahrenheit` (converte da Celsius)
- Setter `fahrenheit` (converte e salva in Celsius)
- Formula: `F = C × 9/5 + 32`

Ereditarietà e riuso

L'**ereditarietà** è un concetto chiave dell'OOP, insieme all'**incapsulamento**.

Si tratta di un meccanismo che consente di definire una nuova classe specializzandone una esistente, ossia "ereditando" i suoi campi e metodi, eventualmente modificandoli o aggiungendone di nuovi.

Si tratta dunque di una strategia di **riuso** di codice già scritto e testato. Inoltre, influenza anche il **polimorfismo** che ne consegue.

Scenari di riuso ed estensione

- Data una classe, realizzarne un'altra con **caratteristiche** solo **in parte diverse** (o nuove)
- Come sopra, ma **senza disporre dei sorgenti** della classe originaria (ad esempio, la classe di partenza è di libreria)
- Data una classe, crearne una **più specializzata** (ad esempio, più robusta e sicura, anche se più lenta)
- Creare **gerarchie** di classi, ossia **di comportamenti**

Ereditarietà con `extends`

Una classe può **specializzarsi** da un'altra usando `extends`:

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

  verso() {
    console.log("Verso generico");
  }
}

class Cane extends Animale {
  verso() {
    console.log("Bau bau!");
  }
}

const fido = new Cane("Fido");
console.log(fido.nome);  // "Fido" (ereditato da Animale)
fido.verso();            // "Bau bau!" (ridefinito in Cane)
```

Concetto: `Cane` **eredita** constructor e metodi di `Animale`, ma può **ridefinire** (override) il comportamento.

`super`: accedere alla classe genitore

Con `super()` nel constructor e `super.metodo()` nei metodi, accediamo alla classe padre:

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

  presentati() {
    console.log("Sono " + this.nome);
  }
}

class Cane extends Animale {
  constructor(nome, razza) {
    super(nome);           // Chiama constructor genitore
    this.razza = razza;
  }

  presentati() {
    super.presentati();    // Chiama metodo del genitore
    console.log("Sono un " + this.razza);
  }
}
```

```
const fido = new Cane("Fido", "Labrador");
fido.presentati();
// Output:
// Sono Fido
// Sono un Labrador
```

> ⚠ **Avviso**
>
> Se la classe figlia ha un `constructor`, **deve** chiamare `super()` prima di usare `this`!

Gerarchia di classi

L'ereditarietà crea **catene di specializzazione**:

```
class Veicolo {
  constructor(marca) {
    this.marca = marca;
  }

  info() {
    return "Veicolo " + this.marca;
  }
}

class Auto extends Veicolo {
  constructor(marca, numPorte) {
    super(marca);
    this.numPorte = numPorte;
  }

  info() {
    return super.info() + " con " + this.numPorte + " porte";
  }
}
```

```
class AutoElettrica extends Auto {
  constructor(marca, numPorte, autonomia) {
    super(marca, numPorte);
    this.autonomia = autonomia;
  }

  info() {
    return super.info() + ", autonomia " + this.autonomia + " km";
  }
}

const tesla = new AutoElettrica("Tesla", 4, 500);
console.log(tesla.info());
// "Veicolo Tesla con 4 porte, autonomia 500 km"
```

Relazione "è un": AutoElettrica **è una** Auto, che **è un** Veicolo.

Composizione vs Ereditarietà

Spesso **composizione** è preferibile a **ereditarietà**:

Ereditarietà (“è un”)

```
class Auto extends Veicolo {
  // Auto "è un" Veicolo
}
```

- Relazione rigida
- Difficile cambiare gerarchia
- Accoppiamento forte

Composizione (“ha un”)

```
class Auto {
  constructor() {
    this.motore = new Motore();
    this.ruote = [new Ruota()];
  }
  // Auto "ha un" Motore
}
```

- Relazione flessibile
- Componenti sostituibili
- Accoppiamento debole

Esercizio: Gerarchia di forme

Proviamo l'ereditarietà

Crea una gerarchia:

- definiamo una classe `Forma`, che sarà la classe base: `constructor(colore)`, metodo `descrivi()`
- definiamo una classe `Rettangolo` che estende `Forma`: aggiunge `larghezza`, `altezza`, metodo `area()`
- definiamo una classe `Quadrato` che estende `Rettangolo`: il costruttore prende solo un parametro `lato`

Crea istanze e testa i metodi ereditati.

Polimorfismo: comportamenti diversi

Il **polimorfismo** permette a oggetti diversi di rispondere allo stesso metodo in modo specifico:

```
class Strumento {
  suona() {
    return "Suono generico";
  }
}

class Chitarra extends Strumento {
  suona() {
    return "Ding ding!";
  }
}

class Piano extends Strumento {
  suona() {
    return "Pam pam!";
  }
}

const strumenti = [new Chitarra(), new Piano(), new Strumento()];
strumenti.forEach(s => console.log(s.suona()));

// Ding ding!
// Pam pam!
// Suono generico
```

Late binding: il metodo corretto viene scelto **a runtime** in base al tipo reale dell'oggetto.

Duck-typing

> Se cammina come un’anatra e starnazza come un’anatra, è un’anatra.

In JavaScript, puoi usare un oggetto **dove serve un comportamento**, senza ereditarietà:

```
// Nessuna classe comune, ma tutti hanno speak()
const persona = { speak() { return "Ciao!"; } };
const robot = { speak() { return "Beep boop!"; } };
const animale = { speak() { return "Verso!"; } };

function chiediDiParlare(essere) {
  console.log(essere.speak());  // Non controlla il tipo!
}

chiediDiParlare(persona);   // "Ciao!"
chiediDiParlare(robot);     // "Beep boop!"
chiediDiParlare(animale);   // "Verso!"
```

- **Vantaggio:** flessibilità massima
- **Svantaggio:** nessun controllo formale (errori a runtime se manca il metodo)

`instanceof`: verificare il tipo

Per controllare se un oggetto appartiene a una classe, usiamo `instanceof`:

```
class Animale {}
class Cane extends Animale {}

const fido = new Cane();

console.log(fido instanceof Cane);      // true
console.log(fido instanceof Animale);   // true (ereditarietà)
console.log(fido instanceof Object);    // true (tutto eredita da Object)

const obj = { nome: "Test" };
console.log(obj instanceof Cane);       // false
```

Utile quando:

- Ricevi dati da API e vuoi validare il tipo
- Vuoi comportamenti diversi in base al tipo dell'oggetto

Prototipi: sotto il cofano

JavaScript usa **prototipi**, non classi classiche. Le classi ES6 sono **syntax sugar**:

```
// Quello che scriviamo
class Persona {
  constructor(nome) {
    this.nome = nome;
  }

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

// È internamente equivalente a
function Persona(nome) {
  this.nome = nome;
}

Persona.prototype.saluta = function() {
  console.log("Ciao, sono " + this.nome);
};
```

Catena dei prototipi: `oggetto` → `Persona.prototype` → `Object.prototype` → `null`

`__proto__` e la catena di prototipi

Ogni oggetto JavaScript ha una **proprietà nascosta** `__proto__` che punta al suo prototipo:

```
const mario = new Persona("Mario");

// mario ha accesso a:
// - proprietà proprie: nome
// - metodi di Persona.prototype: saluta()
// - metodi di Object.prototype: toString(), hasOwnProperty()

console.log(mario.__proto__);         // Persona.prototype
console.log(mario.__proto__.__proto__);  // Object.prototype
console.log(mario.__proto__.__proto__.__proto__);  // null (fine della catena)
```

Quando cerchiamo una proprietà o metodo, JavaScript **cerca lungo la catena** finché non lo trova:

1. **Nell'oggetto stesso?** Se sì, usalo
2. **Nel prototipo?** Se sì, usalo
3. **Nel prototipo del prototipo?** Se sì, usalo
4. **Continua fino a null**

Shadowing: sovrascrivere il comportamento

Se ridefinisci un metodo nella classe figlia, **oscuri** (shadow) il metodo del genitore:

```
class Animale {
  verso() {
    return "Verso generico";
  }
}

class Gatto extends Animale {
  verso() {
    return "Miao!";  // Shadowing: copro il metodo del genitore
  }
}

const micio = new Gatto();
console.log(micio.verso());  // "Miao!" (usato Gatto.verso, non Animale.verso)
```

Nota: con `super.verso()` puoi comunque accedere al metodo del genitore.

```
class Gatto extends Animale {
  verso() {
    const generico = super.verso();  // "Verso generico"
    return generico + " ... Miao!";
  }
}

const micio = new Gatto();
console.log(micio.verso());  // "Verso generico ... Miao!"
```

Ispezionare il prototipo

Per accedere al prototipo, il modo corretto sarebbe usando `Object.getPrototypeOf()` anziché `__proto__`:

```
class Persona {
  saluta() {
    console.log("Ciao");
  }
}

const mario = new Persona();

console.log(Object.getPrototypeOf(mario));  // Persona.prototype
console.log(Object.getPrototypeOf(mario) === Persona.prototype);  // true

// Verifica se un metodo è proprio dell'oggetto o ereditato
console.log(mario.hasOwnProperty("nome"));   // false (se non assegnato)
console.log(mario.hasOwnProperty("saluta")); // false (è ereditato)
console.log("saluta" in mario);              // true (proprio o ereditato)
```

Può essere utile per:

- Debug e ispezione del codice
- Verificare relazioni di ereditarietà

Perché le classi funzionano: riassunto

Ricapitolando, le **classi ES6** sono una sintassi più pulita e accessibile sopra i **prototipi**:

```
// Classe ES6 (quello che usiamo)
class Persona {
  constructor(nome) {
    this.nome = nome;
  }
  saluta() {
    console.log("Ciao, sono " + this.nome);
  }
}

const mario = new Persona("Mario");
```

Internamente, JavaScript:

1. Crea un oggetto vuoto `{}`
2. Imposta `__proto__` = `Persona.prototype`
3. Esegue il `constructor` per inizializzare `this`
4. Ritorna l'oggetto

Quando cerchiamo `mario.saluta()`, JS **segue la catena prototipale** e trova il metodo in `Persona.prototype`.

Esercizio: Ricettario OOP

Ricettario con le classi

Partendo dal ricettario interattivo della lezione precedente, trasformiamolo usando le classi.

1. Definisci una classe `Ricetta` per rappresentare una singola ricetta (nome, ingredienti, preparazione, ecc).
2. Definisci una classe `Ricettario` per gestire la collezione di ricette e la navigazione.
3. Visualizza una ricetta alla volta (aggiorna il DOM in base alla ricetta corrente).
4. Aggiungi i pulsanti “Precedente” e “Successiva” per navigare tra le ricette e mostra l’indice corrente (memorizza l’indice e aggiorna la vista al click).
5. Disabilita i pulsanti ai limiti (prima/ultima ricetta).

Esercizio: Ricettario OOP (esempi)

Esempio di ricette da aggiungere al ricettario

Alla ricetta “Carbonara” già presente nel ricettario, possiamo aggiungere altre due ricette classiche:

Cacio e pepe

Ingredienti

- Spaghetti — 80 g
- Pecorino — 30 g
- Pepe — q.b.

Preparazione

1. Cuoci la pasta.
2. Emulsiona acqua di cottura con il pecorino fino a ottenere una crema.
3. Unisci la pasta e abbonda con pepe.

Amatriciana

Ingredienti

- Bucatini — 80 g
- Guanciale — 37.5 g
- Pomodoro (passata o pelati) — 75 g
- Pecorino — 15 g
- Pepe — q.b.

Preparazione

1. Rosola il guanciale.
2. Aggiungi il pomodoro e fai cuocere la salsa.
3. Unisci la pasta e spolvera con pecorino prima di servire.