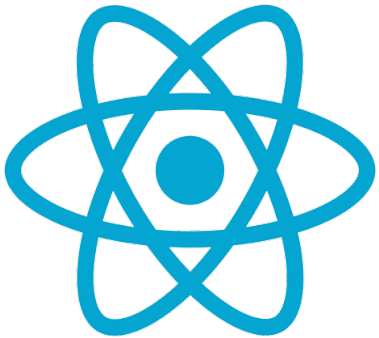
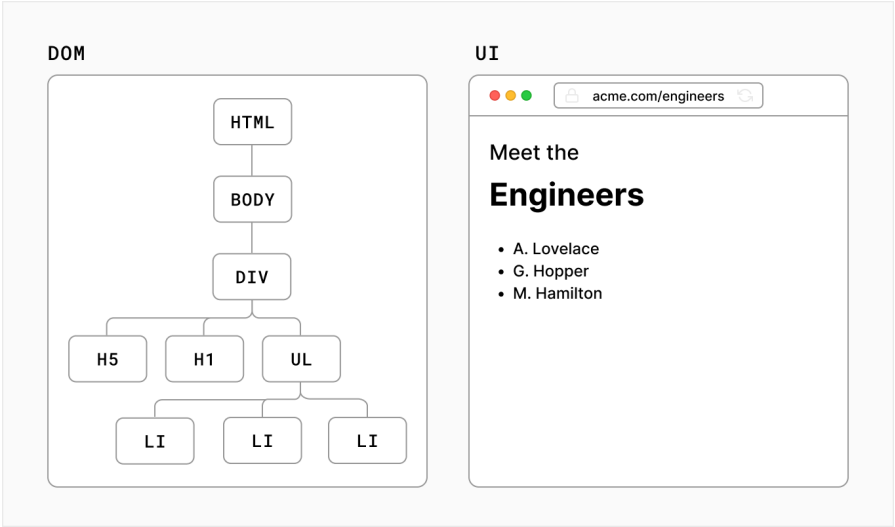
Ripasso JS

Delisio Roberto



Cos'è il DOM?

Il DOM è una rappresentazione di oggetti degli elementi HTML. Funge da ponte tra il codice e l'interfaccia utente e ha una struttura ad albero con relazioni padre e figlio.



Puoi utilizzare metodi DOM e un linguaggio di programmazione, come JavaScript, per ascoltare gli eventi utente e manipolare il DOM selezionando, aggiungendo, aggiornando ed eliminando elementi specifici nell'interfaccia utente. La manipolazione del DOM ti consente non solo di indirizzare elementi specifici, ma anche di cambiarne lo stile e il contenuto.

COS'È JAVASCRIPT

- Originariamente, JavaScript è stato creato per essere eseguito nei browser, ma oggi, grazie a tecnologie come Node.js e Deno, è possibile utilizzarlo anche al di fuori di essi. Tuttavia, il focus di questo corso sarà su React, una libreria frontend basata su browser per costruire interfacce utente.
- È utile sapere che JavaScript non è limitato ai browser. Con tecnologie aggiuntive come Capacitor o React Native, è possibile sviluppare anche applicazioni mobili.
- In questo corso ci concentreremo su JavaScript nel contesto del browser. La sintassi e le regole generali rimangono le stesse, indipendentemente dall'ambiente in cui si scrive il codice.
- Per aggiungere JavaScript a un sito web, ci sono due opzioni principali: inserire il codice direttamente nel file HTML utilizzando il tag <script> o importare file JavaScript esterni. L'approccio inline è sconsigliato per progetti più grandi, poiché può rendere il codice difficile da mantenere. In genere, è meglio mantenere il codice JavaScript in file separati.
- Nel contesto di un progetto React, è raro dover aggiungere manualmente questi tag di script al file HTML. I progetti React utilizzano spesso un processo di compilazione che inietta automaticamente questi tag.
- Nella prossima lezione, esploreremo perché utilizziamo questo processo di compilazione nei progetti React e approfondiremo la sintassi di importazione-esportazione e l'uso dei moduli JavaScript.

IMPORT EXPORT

In questo corso, esploreremo la sintassi di importazione ed esportazione in JavaScript, una pratica essenziale per mantenere il codice manutenibile e gestibile, specialmente in progetti avanzati come quelli basati su React , Vue e Angular.

Immaginiamo di avere un file `util.js` in cui è definita una variabile `API_KEY`. Per rendere questa variabile accessibile in altri file, come `App.js`, dobbiamo esportarla usando la parola chiave `export`.

```
export let API_KEY = "chiave_secreta"; // util.js
```

Successivamente, possiamo importare `API_KEY` in `App.js` utilizzando la parola chiave `import`.

```
import { API_KEY } from './util.js'; // App.js
console.log(API_KEY);
```

Varianti di Esportazione e Importazione

1. Esportazione Predefinita: Se un file esporta un solo elemento, è possibile utilizzare `export default`.

```
export default "chiave_secreta";  // util.js
```

E l'importazione diventa:

```
import miaChiave from './util.js'; // App.js
```

IMPORT EXPORT

2. Importazione Multipla: Se un file esporta più elementi, è possibile importarli tutti insieme.

```
import * as util from './util.js'; // App.js
    console.log(util.API_KEY);
```

3. Alias: È possibile rinominare variabili durante l'importazione usando la parola chiave `as`.

```
import { API_KEY as chiave } from './util.js'; // App.js
```

- In progetti React, Vue e Angular, l'estensione `.js` è spesso omessa durante l'importazione a causa del processo di compilazione.

- L'attributo `type="module"` è necessario quando si lavora con JavaScript vanilla per abilitare la sintassi di importazione ed esportazione.

IMPORT EXPORT

Esportazione Multipla

Se un file esporta più di un elemento, è possibile esportarli tutti insieme. Ad esempio, nel file `util.js` potremmo avere:

```
// util.js

export let API_KEY = "chiave_secreta";

export let ANOTHER_VARIABLE = "un_altro_valore";
```

Per importare più elementi esportati da un singolo file, si utilizza la seguente sintassi:

```
// App.js

import { API_KEY, ANOTHER_VARIABLE } from './util.js';

console.log(API_KEY, ANOTHER_VARIABLE);
```

In questo modo, sia `API_KEY` che `ANOTHER_VARIABLE` saranno disponibili in `App.js`.

IMPORT EXPORT

È possibile importare più esportazioni in un singolo oggetto. Questo è particolarmente utile quando si ha un gran numero di esportazioni da un modulo e si desidera raggrupparle in un unico oggetto. Ecco come si fa:

Supponiamo di avere un file `util.js` con più esportazioni:

```
// util.js

export let API_KEY = "chiave_secreta";

export let ANOTHER_VARIABLE = "un_altro_valore";
```

Ora, nel tuo file `App.js`, puoi importare tutte le esportazioni in un oggetto, ad esempio `util`:

```
// App.js

import * as util from './util.js';

console.log(util.API_KEY);  // Output: "chiave_secreta"

console.log(util.ANOTHER_VARIABLE);  // Output: "un_altro_valore"
```

In questo esempio, tutte le variabili esportate dal file `util.js` sono raggruppate nell'oggetto `util`. Puoi quindi accedere a ciascuna di esse utilizzando la notazione a punti.

VARIABILI

In JavaScript, è possibile gestire un'ampia varietà di valori, come stringhe, numeri, booleani e i valori speciali null e undefined. Questi ultimi indicano semplicemente che una determinata variabile non contiene ancora alcun valore. Esistono anche valori di tipo oggetto, che esamineremo più avanti.

Un valore in JavaScript può essere creato nel momento in cui è necessario. Ad esempio, se si vuole visualizzare "Hello world" nella console, basta creare la stringa con virgolette doppie o singole. Ma spesso è utile memorizzare i valori in variabili per poterli riutilizzare. Le variabili agiscono come contenitori di dati e possono essere create con la parola chiave **let**.

Per esempio, potrei creare una variabile chiamata **userMessage** e memorizzare la stringa "Hello world" al suo interno. Poi, ogni volta che voglio utilizzare quel valore, posso semplicemente fare riferimento al nome della variabile.

Oltre alle variabili, abbiamo anche le costanti, che si creano con la parola chiave const. La differenza fondamentale è che il valore di una costante non può essere modificato una volta assegnato. Personalmente, preferisco usare **const** ogni volta che un valore non deve essere riassegnato, per rendere più chiaro il mio intento.

In sintesi, conoscere la differenza tra let e const e quando utilizzarli è fondamentale per scrivere codice JavaScript efficace e leggibile.

VARIABILI

Utilizzo di `let`

```
let nome = "Mario";

let eta = 30;

let isSviluppatore = true;

// Riassegnazione di valore

nome = "Luigi";
```

Utilizzo di `const`

```
const piGreco = 3.14159;

const giorniNellaSettimana = 7;

// Questo genererà un errore perché non è possibile riassegnare una costante

// piGreco = 3.15;
```

Variabili con tipi diversi

```
let stringa = "Ciao, mondo!";

let numero = 42;

let booleano = true;

let nullo = null;

let indefinito;
```

VARIABILI

Variabili e operazioni

```
let a = 10;
let b = 20;
let somma = a + b;  // somma sarà 30
```

Variabili e stringhe

```
let saluto = "Ciao";
let nomeUtente = "Marco";
let messaggioCompleto = saluto + ", " + nomeUtente + "!";  // "Ciao, Marco!"
```

Variabili in un oggetto

```
let persona = {
  nome: "Anna",
  eta: 25,
  isStudente: true
};
```

Variabili in un array

```
let numeri = [1, 2, 3, 4, 5];
```

VARIABILI

In JavaScript, `let`, `const` e `var` sono tutti utilizzati per dichiarare variabili, ma presentano alcune differenze significative:

`let`

1. **Block Scope**: Le variabili dichiarate con `let` sono limitate allo scope del blocco in cui sono dichiarate, così come a qualsiasi blocco annidato.

2. **Riassegnazione**: È possibile riassegnare nuovi valori a una variabile dichiarata con `let`.

3. **Non Inizializzata**: Una variabile dichiarata con `let` può essere dichiarata senza inizializzazione.

4. **Non può essere dichiarata di nuovo**: Nel medesimo scope, non è possibile dichiarare di nuovo una variabile con lo stesso nome.

```
let x = 10;

if (true) {

  let x = 20;  // Diverso dall'x esterno

}
```

`const`

1. **Block Scope**: Come `let`, anche `const` ha uno scope di blocco.

2. **Non Riassegnabile**: Una volta assegnato un valore a una variabile `const`, non può essere riassegnato.

3. **Deve essere inizializzata**: Una variabile `const` deve essere inizializzata al momento della dichiarazione.

4. **Non può essere dichiarata di nuovo**: Come `let`, anche una variabile `const` non può essere dichiarata di nuovo nello stesso scope.

```
const y = 30;

// y = 40;  // Errore, non può essere riassegnata
```

VARIABILI

`var`

1. **Function Scope**: A differenza di `let` e `const`, `var` è limitato allo scope della funzione in cui è dichiarato, o allo scope globale se dichiarato al di fuori di una funzione.

2. **Riassegnazione**: Come `let`, è possibile riassegnare nuovi valori a una variabile dichiarata con `var`.

3. **Inizializzazione**: Come `let`, una variabile dichiarata con `var` può essere dichiarata senza inizializzazione.

4. **Può essere dichiarata di nuovo**: È possibile dichiarare di nuovo una variabile con lo stesso nome nello stesso scope.

```
var z = 50;

if (true) {

  var z = 100;  // Stesso z, il suo valore viene riassegnato

}
```

L'"hoisting" è un comportamento particolare delle variabili dichiarate con `var` in JavaScript. Quando una variabile viene dichiarata con `var`, la sua dichiarazione viene "sollevata" (o "hoisted") all'inizio del suo scope attuale (che può essere l'intera funzione o, se dichiarata al di fuori di una funzione, l'intero script). Tuttavia, solo la dichiarazione viene sollevata, non l'inizializzazione.

Questo significa che è possibile utilizzare una variabile prima della sua dichiarazione nel codice, ma la variabile esisterà con il valore `undefined` fino al punto in cui viene effettivamente inizializzata.

Ecco un esempio per illustrare l'hoisting con `var`:

```
console.log(a);  // Output: undefined

var a = 5;

console.log(a);  // Output: 5
```

Questo comportamento può portare a risultati inaspettati e potenzialmente a bug, ed è una delle ragioni per cui l'uso di `let` e `const` è generalmente preferito, poiché non presentano questo comportamento di hoisting.

OPERATORI

Sì, `let` e `const` sono molto importanti in JavaScript moderno per una gestione più prevedibile e sicura delle variabili, soprattutto quando si tratta di evitare l'hoisting e di garantire l'immuabilità delle variabili.

Vediamo ora alcuni operatori

- Operatori Aritmetici: Come `+`, `-`, `*`, `/` che possono essere utilizzati per eseguire operazioni matematiche.

```
let somma = 10 + 5;  // 15
let differenza = 10 - 5;  // 5
```

- Concatenazione di Stringhe: L'operatore `+` può anche essere utilizzato per concatenare stringhe.

```
let saluto = "Hello" +" " + "World";  // "Hello World"
```

- Operatori di Confronto: Come `===`, `!==`, `<`, `>`, `<=`, `>=` che sono utilizzati per confrontare valori e restituire un booleano (`true` o `false`).

```
let isEqual = 10 === 10;  // true
let isGreater = 10 > 5;  // true
```

Operatori Logici: Come `&&`, `||`, e `!` che sono utilizzati per eseguire operazioni logiche tra due o più espressioni booleane.

```
let andOperator = true && false;  // false
let orOperator = true || false;  // true
```

- Istruzioni Condizionali: Come `if`, `else if` e `else`, che permettono di eseguire blocchi di codice basati su condizioni.

```
if (10 === 10) {
    console.log("I numeri sono uguali");
} else {
    console.log("I numeri sono diversi");
}
```

FUNZIONI

Oltre alle variabili e alle costanti, un altro elemento fondamentale in JavaScript sono le funzioni. Le funzioni possono essere definite utilizzando la parola chiave `function` o attraverso la sintassi delle funzioni freccia, che esamineremo più avanti.

Definizione di una Funzione

Per definire una funzione, si utilizza la parola chiave `function`, seguita da un nome e da un elenco di parametri racchiusi tra parentesi tonde. Il corpo della funzione è delimitato da parentesi graffe e contiene il codice che verrà eseguito quando la funzione viene chiamata.

```
function saluta(nomeUtente, messaggio) {
  console.log(nomeUtente + ": " + messaggio);
}
```

Chiamata di una Funzione

Una funzione definita non viene eseguita automaticamente. Per eseguirla, è necessario chiamarla utilizzando il suo nome seguito da parentesi tonde.

```
saluta("Max", "Ciao");
saluta("Manuel", "Come va?");
```

Parametri e Valori Predefiniti

Le funzioni possono accettare parametri, che sono variabili utilizzate all'interno della funzione. È anche possibile assegnare valori predefiniti ai parametri.

```
function saluta(nomeUtente = "Utente", messaggio = "Ciao") {
  console.log(nomeUtente + ": " + messaggio);
}
```

FUNZIONI

Valore di Ritorno

Le funzioni possono anche restituire un valore utilizzando la parola chiave `return`.

```
function creaSaluto(nomeUtente, messaggio) {
  return nomeUtente + ": " + messaggio;
}
const saluto1 = creaSaluto("Max", "Ciao");
const saluto2 = creaSaluto("Manuel", "Come va?");
console.log(saluto1);
console.log(saluto2);
```

Nomi delle Funzioni

È importante dare alle funzioni nomi descrittivi che indichino cosa fanno. I nomi delle funzioni dovrebbero seguire le stesse convenzioni utilizzate per le variabili, come la notazione camelCase.

In sintesi, le funzioni sono un concetto cruciale in JavaScript e saranno frequentemente utilizzate, soprattutto quando lavoreremo con Vue. Le funzioni non solo permettono di raggruppare e riutilizzare il codice, ma possono anche accettare parametri e restituire valori, rendendole estremamente flessibili e potenti.

FUNZIONI FRECCIA

Le funzioni freccia sono un'altra sintassi per definire funzioni in JavaScript, introdotte con ES6. Sono particolarmente utili quando si tratta di funzioni anonime, ovvero funzioni che non hanno un nome esplicito.

Sintassi della Funzione Freccia

La sintassi della funzione freccia è più concisa rispetto alla sintassi tradizionale. Si inizia con gli argomenti racchiusi tra parentesi, seguiti da una freccia `=>` e infine il corpo della funzione racchiuso tra parentesi graffe.

```
const saluta = (nomeUtente, messaggio) => {
  console.log(nomeUtente + ": " + messaggio);
};
```

Funzioni Anonime

Le funzioni freccia sono spesso utilizzate come funzioni anonime, specialmente quando si passa una funzione come argomento ad un'altra funzione o come callback.

```
// Utilizzo in un evento onClick in React
<button onClick={() => saluta("Max", "Ciao")}>Clicca qui</button>
```

FUNZIONI FRECCIA

Quando Usare Funzioni Freccia

Le funzioni freccia sono utili quando:

- Si ha bisogno di una funzione anonima.
- Si vuole una sintassi più concisa.
- Non si ha bisogno di un proprio oggetto `this` all'interno della funzione.

Quando Usare la Parola Chiave `function`

La parola chiave `function` è più adatta quando:

- Si ha bisogno di una funzione con un proprio oggetto `this`.
- Si ha bisogno di una funzione con un nome per facilitare il debugging.
- Si sta definendo un costruttore (le funzioni freccia non possono essere usate come costruttori).

Entrambi gli approcci sono validi e verranno utilizzati nel corso. La scelta tra i due dipende dal contesto specifico e dalle vostre preferenze personali. È importante essere a proprio agio con entrambe le sintassi, poiché troverete entrambi gli stili di funzione nel codice JavaScript moderno.

APPROFONDIMENTO SUL THIS

La differenza principale tra le funzioni definite con la parola chiave `function` e le funzioni freccia riguarda il comportamento della parola chiave `this`.

Funzioni Normali (`function`)

Nelle funzioni definite con la parola chiave `function`, la parola chiave `this` è dinamica: il suo valore viene determinato dal modo in cui la funzione viene chiamata. Se una funzione viene chiamata come un metodo di un oggetto, `this` si riferisce all'oggetto stesso. Se la funzione viene chiamata in modo indipendente, `this` si riferisce all'oggetto globale (o è `undefined` in modalità strict).

```
function mostraNome() {
  console.log(this.nome);
}
const persona = {
  nome: "Max",
  mostraNome: mostraNome
};
persona.mostraNome();  // Output: "Max"
const funzioneIsolata = persona.mostraNome;
funzioneIsolata();  // Output: `undefined` o errore in modalità strict
```

APPROFONDIMENTO SUL THIS

Funzioni Freccia (`=>`)

Nelle funzioni freccia, il valore di `this` è determinato dal contesto in cui la funzione è stata definita, non da come viene chiamata. In altre parole, le funzioni freccia ereditano il valore di `this` dal loro scope circostante al momento della definizione.

```
const persona = {
  nome: "Max",
  mostraNome: function() {
    setTimeout(() => {
      console.log(this.nome);
    }, 1000);
  }
};
persona.mostraNome();  // Output: "Max" dopo 1 secondo
```

In questo esempio, la funzione freccia all'interno di `setTimeout` eredita il valore di `this` dal metodo `mostraNome`, permettendo di accedere alla proprietà `nome` dell'oggetto `persona`.

Riassunto

- Le funzioni normali hanno un `this` dinamico che dipende da come vengono chiamate.

- Le funzioni freccia hanno un `this` lessicale che dipende da dove vengono definite.

Scegliere tra i due tipi di funzioni dipende dal comportamento desiderato per `this` nel tuo codice.

OGGETTI

Ora che abbiamo esplorato le funzioni, torniamo a parlare dei valori, e in particolare degli oggetti. Come dovreste già sapere, gli oggetti in JavaScript servono per raggruppare più valori sotto un'unica entità. Ad esempio, se ho un nome utente, che potrebbe essere "Rob", e un'età, che potrebbe essere 34, posso raggruppare questi dati in un oggetto "utente".

```
const utente = {
  nome: Rob',
  eta: 46
};
```

Questo oggetto può essere stampato nella console per l'ispezione, e posso anche accedere ai suoi singoli campi usando la notazione a punti, come `utente.nome`.

Oltre a contenere semplici coppie chiave-valore, gli oggetti possono anche avere funzioni, chiamate metodi. Ad esempio:

```
const utente = {
  nome: ‘Rob',
  eta: 46,
  saluta: function() {
    console.log('Ciao!’);
    console.log(this.nome);
  }
};
```

Posso chiamare questo metodo con `utente.saluta()`. All'interno di un metodo, posso usare la parola chiave `this` per accedere ad altre proprietà dell'oggetto.

OGGETTI

Un altro modo per creare oggetti è utilizzare le classi. Una classe funge da "progetto" per creare oggetti.

```
class Utente {
  constructor(nome, eta) {
    this.nome = nome;
    this.eta = eta;
  }
  saluta() {
    console.log('Ciao! ‘ + this.nome);
  }
}


const utenteUno = new Utente('Manuel', 35);
```

Qui, `utenteUno` è un'istanza della classe `Utente` e ha accesso al metodo `saluta`.

In questo corso, non ci concentreremo troppo sull'uso delle classi, ma è un concetto utile da conoscere. Ora possiamo passare al prossimo argomento.

ARRAY E METODI DEGLI ARRAY

Oltre agli oggetti, un altro elemento fondamentale in JavaScript sono gli array. Anche se tecnicamente sono oggetti, rappresentano una categoria speciale.

Si creano utilizzando parentesi quadre aperte e chiuse. A differenza degli oggetti, che utilizzano coppie chiave-valore, gli array memorizzano solo valori in un ordine specifico, accessibili tramite il loro indice.

Ad esempio, se ho una lista di hobby come sport, cucina e lettura, posso accedere a questi valori utilizzando l'indice, che parte da zero. Gli array sono molto comuni in JavaScript perché spesso è necessario memorizzare liste di valori. Possono contenere qualsiasi tipo di valore, inclusi altri array e oggetti.

Gli array offrono vari metodi utili. Ad esempio, il metodo `push` aggiunge un nuovo elemento all'array. Un altro metodo utile è `findIndex`, che trova l'indice di un determinato valore. Questo metodo accetta una funzione come input, che viene eseguita per ogni elemento dell'array.

Un altro metodo frequentemente utilizzato è `map`, che trasforma ogni elemento dell'array in un altro elemento. Come `findIndex`, anche `map` accetta una funzione come input. Questa funzione viene eseguita per ogni elemento dell'array, e il valore restituito diventa il nuovo elemento dell'array.

In sintesi, gli array sono strumenti potenti per memorizzare e manipolare liste di valori in JavaScript. Offrono una varietà di metodi utili per la manipolazione e l'accesso ai dati, rendendoli strumenti indispensabili per qualsiasi sviluppatore JavaScript.

ARRAY E METODI DEGLI ARRAY

Certo, ecco alcuni esempi di codice basati sul testo che hai fornito:

Creazione di un array

```
const hobbies = ['sport', 'cucina', 'lettura'];
```

Accesso a un elemento dell'array tramite indice

```
console.log(hobbies[0]);  // Output: "sport"
```

Utilizzo del metodo `push` per aggiungere un elemento

```
hobbies.push('lavoro');
console.log(hobbies);  // Output: ["sport", "cucina", "lettura", "lavoro"]
```

Utilizzo del metodo `findIndex` per trovare l'indice di un elemento

```
const index = hobbies.findIndex(item => item === 'sport');
console.log(index);  // Output: 0
```

Utilizzo del metodo `map` per trasformare gli elementi

```
const modifiedHobbies = hobbies.map(item => item + '!');
console.log(modifiedHobbies);  // Output: ["sport!", "cucina!", "lettura!", "lavoro!"]
```

Utilizzo del metodo `map` per trasformare gli elementi in oggetti

```
const objectHobbies = hobbies.map(item => ({ text: item }));
console.log(objectHobbies);
// Output: [{ text: "sport" }, { text: "cucina" }, { text: "lettura" }, { text: "lavoro" }]
```

Spero che questi esempi ti siano utili per comprendere meglio come lavorare con gli array in JavaScript.

ASSEGNAZIONE PER RIFERIMENTO

In JavaScript, gli oggetti e gli array sono assegnati per riferimento. Questo significa che quando assegni un oggetto o un array a una nuova variabile, entrambe le variabili puntano allo stesso oggetto o array in memoria. Qualsiasi modifica apportata all'oggetto o all'array attraverso una delle variabili si rifletterà anche sull'altra.

Esempio con oggetti

```
const persona1 = { nome: 'Mario', eta: 30 };
const persona2 = persona1;

persona2.nome = 'Luigi';

console.log(persona1.nome);  // Output: "Luigi"
console.log(persona2.nome);  // Output: "Luigi"
```

In questo esempio, `persona1` e `persona2` puntano allo stesso oggetto. Quando cambiamo il nome attraverso `persona2`, il nome cambia anche per `persona1`.

ASSEGNAZIONE PER RIFERIMENTO

Esempio con array

```
const array1 = [1, 2, 3];
const array2 = array1;
array2.push(4);
console.log(array1);  // Output: [1, 2, 3, 4]
console.log(array2);  // Output: [1, 2, 3, 4]
```

Anche in questo caso, `array1` e `array2` puntano allo stesso array in memoria. Quando aggiungiamo un elemento all'`array2`, l'`array1` viene modificato di conseguenza.

Come evitare l'assegnazione per riferimento

Se vuoi evitare questo comportamento e creare una copia indipendente dell'oggetto o dell'array, puoi utilizzare tecniche come la "shallow copy" con il metodo `Object.assign()` per gli oggetti o lo spread operator (`...`) per gli array.

Esempio di copia di un oggetto

```
const persona3 = Object.assign({}, persona1);
```

Esempio di copia di un array

```
const array3 = [...array1];
```

DESTRUTTURAZIONE

Ci sono due caratteristiche moderne e cruciali di JavaScript che dovreste conoscere, poiché le incontrerete spesso durante il corso. La prima è la destrutturazione di array e oggetti.

Immaginiamo di avere un array che contiene dati relativi al nome utente, come il nome e il cognome. Potremmo voler lavorare con entrambi nel nostro codice. Una soluzione semplice sarebbe creare nuove costanti o variabili, come `firstname` e `lastname`, e assegnarvi i valori dall'array utilizzando gli indici appropriati. Tuttavia, questo approccio può essere semplificato utilizzando la destrutturazione.

Con la destrutturazione, possiamo creare queste due costanti in un unico passaggio. Basta utilizzare le parentesi quadre a sinistra del segno di uguale. Questa sintassi ci permette di estrarre i valori dall'array e assegnarli a nuove variabili in modo più conciso.

La destrutturazione non è limitata agli array; può essere utilizzata anche con gli oggetti. Supponiamo di avere un oggetto `user` con campi come `name` e `age`. Anche in questo caso, potremmo voler estrarre questi valori in costanti o variabili separate. Invece di farlo manualmente, possiamo utilizzare la destrutturazione con parentesi graffe.

È importante notare che, mentre con la destrutturazione di array possiamo scegliere i nomi delle variabili, con gli oggetti dobbiamo utilizzare i nomi delle proprietà esistenti. Tuttavia, è possibile assegnare un alias a queste proprietà utilizzando i due punti.

In sintesi, la destrutturazione è una caratteristica potente di JavaScript che verrà spesso utilizzata nel corso. Ora che la conoscete, sarete meglio preparati per affrontare esempi più complessi.

DESTRUTTURAZIONE

Certamente, ecco alcuni esempi di codice basati sul testo che hai fornito:

Esempio 1: Destrutturazione di un array

```
// Array con nome e cognome
const userData = ['Rob', 'Del'];
// Metodo tradizionale per estrarre i dati
const firstnameOld = userData[0];
const lastnameOld = userData[1];
// Utilizzo della destrutturazione per estrarre i dati
const [firstname, lastname] = userData;
console.log(firstname);  // Output: Rob
console.log(lastname);   // Output: Del
```

Esempio 2: Destrutturazione di un oggetto

```
// Oggetto con nome e età
const user = {
  name: 'Rob',
  age: 46
};
// Metodo tradizionale per estrarre i dati
const nameOld = user.name;
const ageOld = user.age;

// Utilizzo della destrutturazione per estrarre i dati
const { children } = props;

console.log(name);  // Output: Rob
console.log(age);   // Output: 46
```

DESTRUTTURAZIONE

Esempio 3: Destrutturazione con alias

```
// Utilizzo della destrutturazione con alias
const { name: userName, age: userAge } = user;

console.log(userName);  // Output: Max
console.log(userAge);   // Output: 30
```

DESTRUTTURAZIONE DI PARAMETRI DELLE FUNZIONI

La sintassi di destrutturazione spiegata nella lezione precedente può essere utilizzata anche nelle liste di parametri delle funzioni.

Ad esempio, se una funzione accetta un parametro che conterrà un oggetto, questo può essere destrutturato per "estrarre" le proprietà dell'oggetto e renderle disponibili come variabili con ambito locale (cioè, variabili disponibili solo all'interno del corpo della funzione).

Ecco un esempio:

```
function storeOrder(order) {
  localStorage.setItem('id', order.id);
  localStorage.setItem('currency', order.currency);
}
```

Invece di accedere alle proprietà dell'ordine tramite la "notazione a punto" all'interno del corpo della funzione `storeOrder`, potresti utilizzare la destrutturazione in questo modo:

```
function storeOrder({id, currency}) { // destrutturazione
  localStorage.setItem('id', id);
  localStorage.setItem('currency', currency);
}
```

DESTRUTTURAZIONE DI PARAMETRI DELLE FUNZIONI

La sintassi di destrutturazione è la stessa insegnata nella lezione precedente, solo che non è necessario creare manualmente una costante o una variabile.

Invece, `id` e `currency` vengono "estratti" dall'oggetto in arrivo (cioè, l'oggetto passato come argomento a `storeOrder`).

È molto importante capire che `storeOrder` accetta ancora un solo parametro in questo esempio! Non accetta due parametri. Invece, è un singolo parametro, un oggetto che poi viene semplicemente destrutturato internamente.

La funzione verrebbe ancora chiamata in questo modo:

storeOrder({id: 5, currency: 'USD', amount: 15.99}); // un solo argomento/valore!

SPRED OPERATOR

Ora, un altro concetto fondamentale che dovreste conoscere riguarda l'operatore di spread in JavaScript. Supponiamo, ad esempio, di avere un elenco di hobby e di volerlo unire con un altro elenco di hobby. In questo caso, potremmo avere un altro elenco che contiene un solo elemento.

Per creare un elenco unito, potrei utilizzare l'operatore di spread, rappresentato da tre punti, seguito dal nome dell'array che voglio unire al nuovo array. Ad esempio, se ho un array chiamato "hobby", posso utilizzare i tre punti per estrarre tutti gli elementi di questo array e aggiungerli al nuovo array.

Se utilizzassi la sintassi senza l'operatore di spread, finirei con un array annidato, che potrebbe non essere ciò che voglio. Ma usando l'operatore di spread, i valori vengono estratti dagli array originali e aggiunti come elementi separati nel nuovo array.

Questo operatore di spread è molto utile e lo useremo di tanto in tanto in questo corso. È importante notare che l'operatore di spread può essere utilizzato non solo con gli array, ma anche con gli oggetti.

Per esempio, se ho un oggetto "utenteEsteso" che contiene una proprietà "isAdmin", e voglio unire le proprietà di un altro oggetto "utente", posso utilizzare l'operatore di spread. Questo estrarrebbe tutte le coppie chiave-valore dall'oggetto "utente" e le aggiungerebbe all'oggetto "utenteEsteso".

In sintesi, l'operatore di spread è un potente strumento per estrarre elementi da array e proprietà da oggetti, permettendoci di unirli in nuovi array o oggetti.

SPRED OPERATOR

Esempio 1: Utilizzo dell'operatore di spread con array

```
// Array iniziali
const hobby1 = ['Nuoto', 'Ciclismo'];
const hobby2 = ['Lettura'];

// Unione degli array senza l'operatore di spread
const unioneSenzaSpread = [hobby1, hobby2];
console.log(unioneSenzaSpread); // Output: [['Nuoto', 'Ciclismo'], ['Lettura']]

// Unione degli array con l'operatore di spread
const unioneConSpread = [...hobby1, ...hobby2];
console.log(unioneConSpread); // Output: ['Nuoto', 'Ciclismo', 'Lettura']
```

Esempio 2: Utilizzo dell'operatore di spread con oggetti

```
// Oggetti iniziali
const utente = { nome: 'Mario', eta: 30 };
const utenteEsteso = { isAdmin: true };

// Unione degli oggetti con l'operatore di spread
const utenteFinale = { ...utente, ...utenteEsteso };
console.log(utenteFinale); // Output: { nome: 'Mario', eta: 30, isAdmin: true }
```

STRUTTURE DI CONTROLLO

Quindi, abbiamo rivisto abbastanza sugli array e gli oggetti. Ora passiamo alle strutture di controllo. Ho già introdotto la parola chiave `if`, che serve per creare le cosiddette istruzioni condizionali `if`. L'obiettivo è confrontare i valori e eseguire il codice all'interno del blocco `if` solo se la condizione specificata è vera.

L'istruzione `if` permette anche di aggiungere un blocco `else` per eseguire del codice nel caso in cui la condizione non sia soddisfatta. È possibile anche utilizzare `else if` per verificare altre condizioni se la prima non è vera. Puoi avere quanti `else if` vuoi, ma solo un blocco `else`.

Generalmente, le istruzioni `if` sono utilizzate per controllare dati che non sono noti in anticipo. Ad esempio, potrei utilizzare la funzione `prompt` del browser per chiedere all'utente di inserire una password. A seconda della password inserita, posso eseguire diversi blocchi di codice.

Un'altra struttura di controllo fondamentale è il ciclo `for`. JavaScript offre diversi tipi di cicli `for`, e uno molto importante che tratteremo in questo corso è il ciclo `for...of`, utilizzato per iterare attraverso un array. Ad esempio, se ho un array di hobby come ['Sport', 'Cucina'], posso utilizzare un ciclo `for...of` per eseguire un blocco di codice per ogni elemento dell'array.

Queste sono le basi delle strutture di controllo in JavaScript, e le vedremo spesso nel corso di questo corso.

STRUTTURE DI CONTROLLO

Esempio 1: Uso di `if`, `else if` e `else`

```
let password = prompt("Inserisci la tua password:");
if (password === "Ciao") {
  console.log("Accesso concesso.");
} else if (password === "ciao") {
  console.log("Accesso concesso, ma attenzione alle maiuscole.");
} else {
  console.log("Accesso negato.");
}
```

Esempio 2: Ciclo `for...of` per iterare un array

```
const hobbies = ["Sport", "Cucina"];

for (const hobby of hobbies) {
  console.log(`Il mio hobby è: ${hobby}`);
}
```

In questo esempio, il ciclo `for...of` passa attraverso ogni elemento dell'array `hobbies` e stampa un messaggio sulla console per ciascuno di essi.

STRUTTURE DI CONTROLLO

Nel contesto di un ciclo `for...of`, la parola chiave `const` può essere utilizzata per dichiarare una variabile che sarà costante all'interno di ogni singola iterazione del ciclo. In altre parole, per ogni iterazione del ciclo, una nuova variabile `const` viene creata e inizializzata con il valore corrente dell'elemento dell'array (o di qualsiasi altro oggetto iterabile).

Ecco un esempio per chiarire:

```
const numeri = [1, 2, 3, 4, 5];


for (const numero of numeri) {
  console.log(numero);
  // 'numero' è costante all'interno di questa iterazione specifica
  // e non può essere modificato.
  // Tuttavia, una nuova 'const numero' sarà creata per la prossima iterazione.
}
```

In questo esempio, la variabile `numero` è dichiarata come `const`, il che significa che non può essere modificata all'interno del blocco del ciclo `for...of`. Tuttavia, per ogni nuova iterazione del ciclo, una nuova istanza di `numero` viene creata, permettendo così di assegnarle il valore corrente dell'elemento dell'array `numeri`.

Quindi, in breve, `const` è utilizzabile in un ciclo `for...of` perché una nuova variabile viene creata per ogni iterazione del ciclo.

FUNZIONI COME PARAMETRI

Con questo, abbiamo quasi concluso questa sezione. Tuttavia, ci sono alcune funzioni fondamentali e concetti avanzati che desidero ripassare qui, così da non avere dubbi quando li incontrerete più avanti nel corso.

Una delle prime caratteristiche di JavaScript che dovete conoscere è la capacità di passare funzioni come argomenti ad altre funzioni. Ad esempio, possiamo utilizzare la funzione `setTimeout`, fornita dal browser, per impostare un timer. Questa funzione accetta due parametri: il primo è una funzione, che può essere definita sia con la parola chiave `function` sia come funzione freccia.

È importante capire che stiamo creando una nuova funzione anonima, poiché non ha un nome specifico. Potremmo anche definirla separatamente con un nome, come `handleTimeout`, e poi passarla a `setTimeout`.

Quando passiamo una funzione come argomento, dobbiamo farlo utilizzando solo il suo nome, senza parentesi. Questo perché vogliamo passare la funzione stessa, non il suo valore di ritorno.

`setTimeout` accetta anche un secondo parametro, che è un numero rappresentante il tempo di attesa in millisecondi prima dell'esecuzione della funzione.

Voglio sottolineare che la capacità di passare funzioni come argomenti non è limitata alle funzioni integrate come `setTimeout`. Potete creare le vostre funzioni personalizzate che accettano altre funzioni come argomenti. Ad esempio, una funzione `greeter` potrebbe accettare una funzione `greet` come parametro e poi eseguirla.

In sintesi, passare funzioni come argomenti è un concetto fondamentale in JavaScript e lo vedremo spesso nel corso. Pertanto, è importante essere a proprio agio con questo concetto.

FUNZIONI COME PARAMETRI

Esempio 1: Utilizzo di `setTimeout` con una funzione anonima

```
setTimeout(function() {
  console.log("Sono passati 3 secondi!");
}, 3000);
```

Esempio 2: Utilizzo di `setTimeout` con una funzione predefinita

```
function handleTimeout() {
  console.log("Sono passati 3 secondi!");
}
setTimeout(handleTimeout, 3000);
```

Esempio 3: Utilizzo di `setTimeout` con una funzione freccia

```
setTimeout(() => {
  console.log("Sono passati 3 secondi!");
}, 3000);
```

Esempio 4: Creazione di una funzione che accetta un'altra funzione come argomento

```
function greeter(greetFn) {
  console.log("Inizio saluto...");
  greetFn();
  console.log("Fine saluto.");
}
// Utilizzo della funzione `greeter`
greeter(() => {
  console.log("Ciao a tutti!");
});
```

FUNZIONI COME PARAMETRI

Il Problema del "Callback Hell"

Quando abbiamo molte operazioni asincrone in sequenza, il codice può diventare difficile da leggere:

```
setTimeout(function() {
    console.log("Prima operazione");
    setTimeout(function() {
        console.log("Seconda operazione");
        setTimeout(function() {
            console.log("Terza operazione");
            // ... e così via
        }, 1000);
    }, 1000);
}, 1000);
```

Questo codice funziona, ma diventa rapidamente illeggibile e difficile da mantenere.

Un Esempio Reale con Richieste

```
// Simulazione di richieste al server
function ottieniUtente(id, callback) {
    setTimeout(() => {
        callback({ id: id, nome: 'Mario' });
    }, 1000);
}

function ottieniOrdini(utente, callback) {
    setTimeout(() => {
        callback(['Ordine 1', 'Ordine 2']);
    }, 1000);
}

// Utilizzo
ottieniUtente(123, function(utente) {
    console.log('Utente ottenuto:', utente.nome);
    ottieniOrdini(utente, function(ordini) {
        console.log('Ordini ottenuti:', ordini);
    });
});
```

FUNZIONI COME PARAMETRI

Ora, un altro aspetto delle funzioni che verrà spesso affrontato in questo corso è la possibilità di definire funzioni all'interno di altre funzioni. Mentre in JavaScript puro questa pratica potrebbe non sembrare immediatamente utile, nel contesto di React diventa molto più rilevante, come vedremo in seguito.

Ad esempio, potremmo avere una funzione chiamata `init` che contiene al suo interno un'altra funzione, magari chiamata `greet`, che esegue un `console.log`. All'interno della funzione `init`, è possibile chiamare `greet`. Tuttavia, non è possibile chiamare `greet` al di fuori di `init`, in quanto `greet` è definita all'interno di `init` e quindi ha uno scope limitato a quella funzione.

In altre parole, `greet` è disponibile solo come variabile all'interno dello scope di `init` e non può essere accessibile al di fuori di esso. D'altro canto, `init` può essere chiamata liberamente, poiché ha uno scope che copre l'intero file, non essendo annidata in un'altra funzione.

Quindi, se eseguite questo codice, vedrete un output nel console. Questo perché alla fine viene eseguita la funzione `init`, che a sua volta chiama internamente `greet`.

In sintesi, è possibile definire ed eseguire funzioni all'interno di altre funzioni. Anche se questa non è una pratica comune in JavaScript puro, diventa molto più comune quando si lavora con React, come vedremo nel corso di questo corso.

FUNZIONI COME PARAMETRI

```
function init() {
  console.log("Funzione init eseguita");

function greet() {
    console.log("Ciao dal greet interno!");
  }

greet();
}
```

```
// Esecuzione della funzione esterna 'init'
init();

// Tentativo di esecuzione della funzione interna 'greet' al di fuori di 'init'
// Questo solleverà un errore perché 'greet' è definita solo all'interno di 'init'
// greet();  // Uncommenting this line will throw an error
```

In questo esempio, abbiamo una funzione `init` che contiene una funzione interna `greet`. All'interno di `init`, chiamiamo `greet`, e tutto funziona come previsto. Tuttavia, se proviamo a chiamare `greet` al di fuori di `init`, otterremo un errore, perché `greet` è definita solo all'interno dello scope di `init`.

INTRODUZIONE ALLE PROMISE

Le **Promise** sono state introdotte per risolvere i problemi delle callback, rendendo il codice più leggibile e gestibile.

Una Promise rappresenta un'operazione che si completerà in futuro e può avere tre stati:

Pending (in attesa): L'operazione non è ancora completata

Fulfilled (risolta): L'operazione è completata con successo

Rejected (rifiutata): L'operazione è fallita

```
const miaPromise = new Promise(function(resolve, reject) {
    setTimeout(() => {
        const successo = true;

        if (successo) {
            resolve('Operazione completata!'); // Successo
        } else {
            reject('Qualcosa è andato storto!'); // Errore
        }
    }, 2000);
});
```

Consumare una Promise con .then() e .catch()

```
miaPromise
    .then(function(risultato) {
        console.log('Successo:', risultato);
    })
    .catch(function(errore) {
        console.log('Errore:', errore);
    });
```

INTRODUZIONE ALLE PROMISE

```
function caricaDatiUtente(id) {
    return new Promise((resolve, reject) => {
        // Simuliamo una richiesta al server
        setTimeout(() => {
            if (id > 0) {
                resolve({
                    id: id,
                    nome: 'Mario Rossi',
                    email: 'mario@email.com'
                });
            } else {
                reject('ID utente non valido');
            }
        }, 1500);
    });
}
```

```
// Utilizzo
caricaDatiUtente(123)
    .then(utente => {
        console.log('Dati utente:', utente);
        console.log('Nome:', utente.nome);
    })
    .catch(errore => {
        console.log('Si è verificato un errore:', errore);
    });
```

INTRODUZIONE ALLE PROMISE

```
caricaDatiUtente(123)
    .then(utente => {
        console.log('Utente caricato:', utente.nome);
        return caricaOrdiniUtente(utente.id); // Ritorna un'altra Promise
    })
    .then(ordini => {
        console.log('Ordini caricati:', ordini);
        return calcolaTotale(ordini); // Ritorna un'altra Promise
    })
    .then(totale => {
        console.log('Totale spesa:', totale);
    })
    .catch(errore => {
        console.log('Errore in una delle operazioni:', errore);
```

ASYNC/AWAIT - SEMPLIFICHIAMO LE PROMISE

Le parole chiave **async** e **await** rendono il codice asincrono più facile da leggere e scrivere, facendolo sembrare codice sincrono.

```
async function miaFunzioneAsincrona() {
    try {
        const risultato = await miaPromise;
        console.log(risultato);
    } catch (errore) {
        console.log('Errore:', errore);
    }
}
```

ASYNC/AWAIT - SEMPLIFICHIAMO LE PROMISE

Con Promise:

```
javascriptfunction ottieniDatiUtente() {
    caricaDatiUtente(123)
        .then(utente => {
            console.log('Utente:', utente);
            return caricaOrdiniUtente(utente.id);
        })
        .then(ordini => {
            console.log('Ordini:', ordini);
        })
        .catch(errore => {
            console.log('Errore:', errore);
        });
}
```

Con Async/Await:

```
javascriptasync function ottieniDatiUtente() {
    try {
        const utente = await caricaDatiUtente(123);
        console.log('Utente:', utente);

        const ordini = await
caricaOrdiniUtente(utente.id);
        console.log('Ordini:', ordini);
    } catch (errore) {
        console.log('Errore:', errore);
    }
}
```

ASYNC/AWAIT - SEMPLIFICHIAMO LE PROMISE

```
javascriptasync function caricaDatiCompleti() {
    try {
        console.log('Inizio caricamento...');
        const utente = await caricaDatiUtente(123);
        console.log('✓ Utente caricato:', utente.nome);

        // Aspetta che gli ordini siano caricati
        const ordini = await caricaOrdiniUtente(utente.id);
        console.log('✓ Ordini caricati:', ordini.length, 'ordini');

        // Aspetta che il totale sia calcolato
        const totale = await calcolaTotale(ordini);
        console.log('✓ Totale calcolato:', totale, '€');

        return { utente, ordini, totale };
    } catch (errore) {
        console.log('Errore durante il caricamento:', errore);
    }
}
caricaDatiCompleti();
```

Caricamento Parallelo

```
javascriptasync function caricaDatiParalleli() {
    try {
        console.log('Caricamento parallelo...');
        const promiseUtente = caricaDatiUtente(123);
        const promiseImpostazioni = caricaImpostazioni();
        const promiseNotifiche = caricaNotifiche();
        // Aspetta che tutte siano completate
        const [utente, impostazioni, notifiche] = await Promise.all([
            promiseUtente,
            promiseImpostazioni,
            promiseNotifiche
        ]);
        console.log('Tutto caricato!');
        return { utente, impostazioni, notifiche };
    } catch (errore) {
        console.log('Errore nel caricamento parallelo:', errore);
    }
}
```