

---

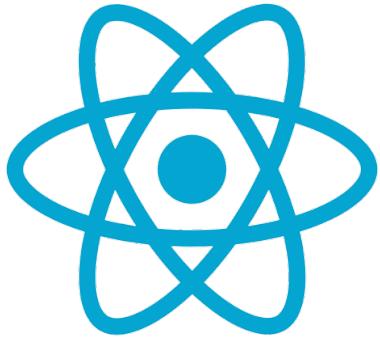
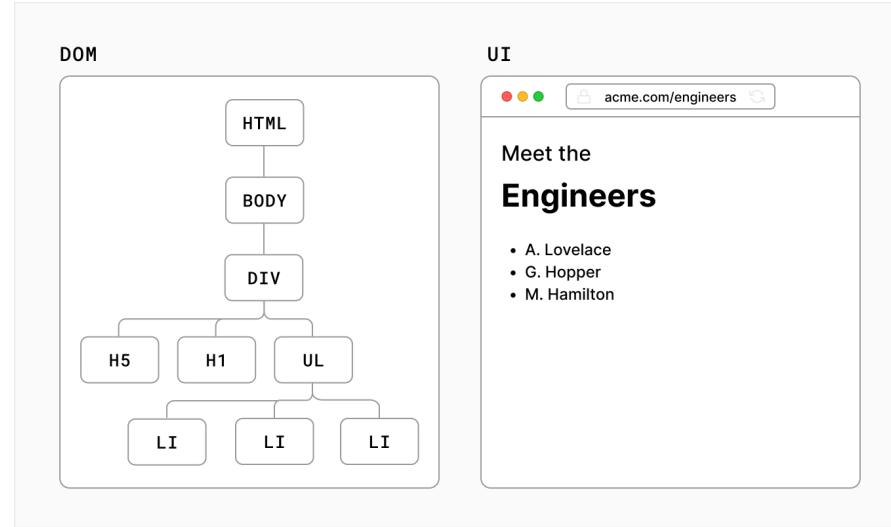
# Ripasso JS

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

I. 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

Si, `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 funzionelsolata = persona.mostraNome;  
  
funzionelsolata(); // 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 personal1 = { nome: 'Mario', eta: 30 };
const persona2 = personal1;

persona2.nome = 'Luigi';

console.log(personal1.nome); // Output: "Luigi"
console.log(persona2.nome); // Output: "Luigi"
```

In questo esempio, `personal1` e `persona2` puntano allo stesso oggetto. Quando cambiamo il nome attraverso `persona2`, il nome cambia anche per `personal1`.

# 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];
```

# DESTRUZIONE

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.

# DESTRUZIONE

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
```

## DESTRUZIONE

### Esempio 3: Destruzione con alias

```
// Utilizzo della destruzione con alias  
const { name: userName, age: userAge } = user;  
  
console.log(userName); // Output: Max  
console.log(userAge); // Output: 30
```

# DESTRUZIONE 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);  
}
```

# DESTRUZIONE 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(resultato) {  
    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();
```

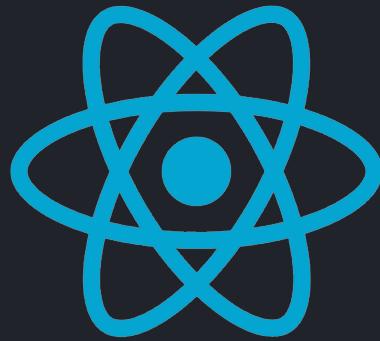
Caricamento Parallelo

```
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);  
  }  
}
```

---

# Oggi parliamo di...

# React Js



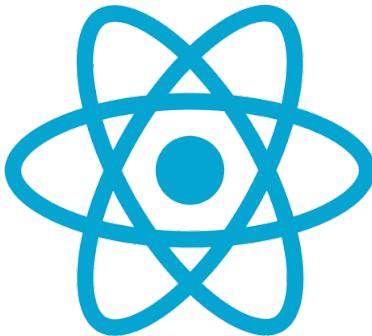
Delisio Roberto

---

---

# Risorse utili e prerequisiti

Vediamo alcune risorse online che possono risultare particolarmente utili:



- Node js
- Chrome
- Visual Studio Code
- [HTMLto JSX Compiler](#)
- <https://codepen.io/>

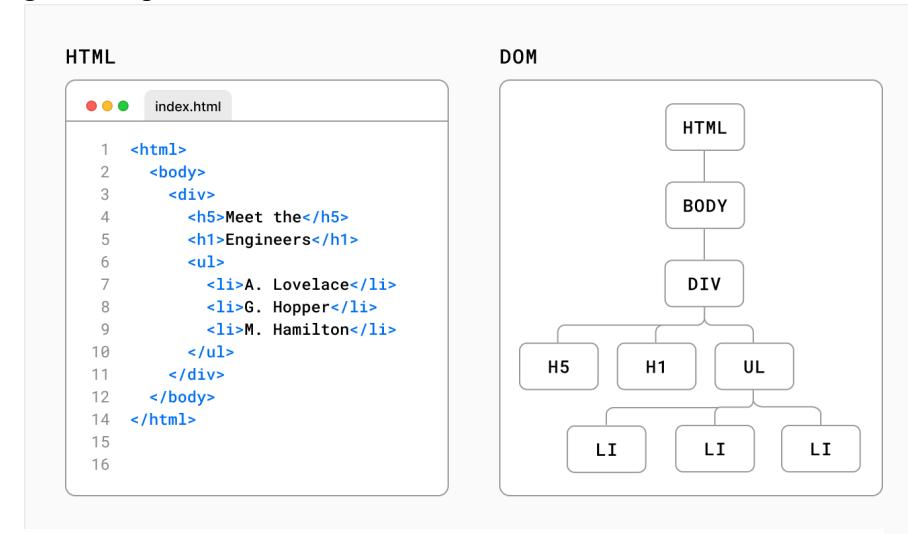
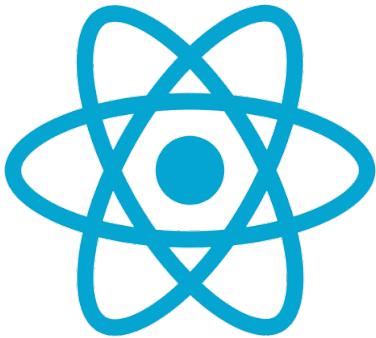
Cosa dobbiamo conoscere

- HTML
- CSS
- Javascript
- Bootstrap

# Come Funziona il DOM

Per capire come funziona React, abbiamo prima bisogno di una conoscenza di base di come i browser interpretano il tuo codice per creare interfacce utente interattive (UI).

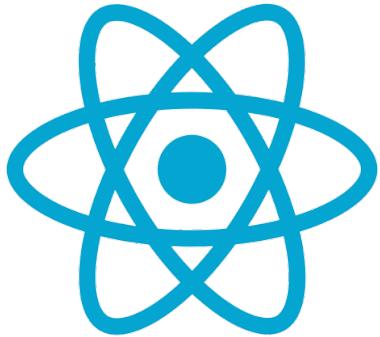
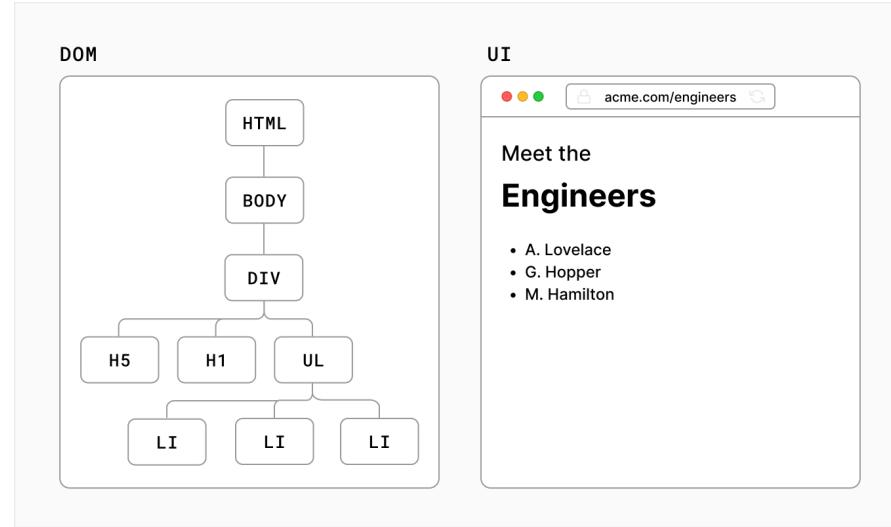
Quando un utente visita una pagina Web, il server restituisce al browser un file HTML che potrebbe assomigliare a questo:



Il browser legge quindi l'HTML e costruisce il **Document Object Model (DOM)**.

# Cos'è il DOM?

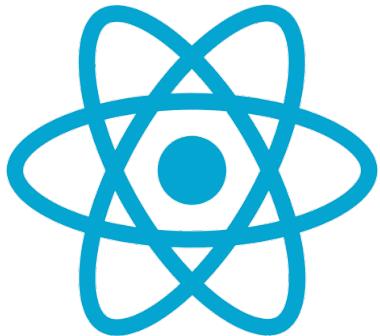
Il DOM è una rappresentazione di oggetti degli elementi HTML. Funge da ponte tra il codice e l'interfaccia utente e ha una struttura ad albero con relazioni padre e figlio.



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 REACTJS

React è una **libreria JavaScript** per la creazione di interfacce utente (UI, User Interface). Sviluppata nel 2013 all'interno di Facebook, adesso React è una libreria open-source supportata da una grande community di programmatore.



React consente di sviluppare applicazioni dinamiche che non necessitano di ricaricare la pagina per visualizzare i dati modificati. Inoltre nelle applicazioni React le modifiche effettuate sul codice si possono visualizzare in tempo reale, permettendo uno sviluppo rapido, efficiente e flessibile delle applicazioni web.

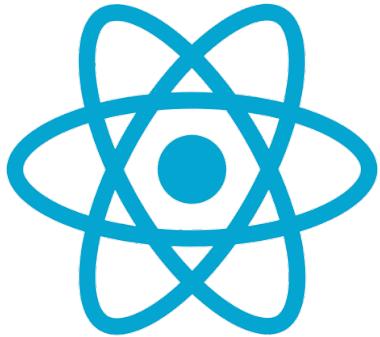
## React è la “V” di MVC

Se dovessimo inquadrare React all'interno di un paradigma, potremmo dire che esso è la “V” di MVC: nasce per gestire la parte relativa alle view, ossia alla presentazione, e all'intercettazione degli eventi di input dell'utente, senza forzare l'adozione di specifiche funzioni né limitare l'interfacciamento ad altre librerie per quanto riguarda l'eventuale comunicazione con un server di backend, o specifiche architetture di binding ai dati, frangenti in cui lo sviluppatore ha libera scelta sugli strumenti con cui integrare React.

---

# Perché usare REACTJS

React è probabilmente la prima libreria JavaScript che nasce con una vocazione specifica: diventare la soluzione definitiva per sviluppatori front-end e app mobile basate su HTML5, la proverbiale "panacea per tutti i mali", il tool che permetta di costruire interfacce utente dinamiche e sempre più complesse rimanendo comunque semplice e intuitivo da utilizzare.



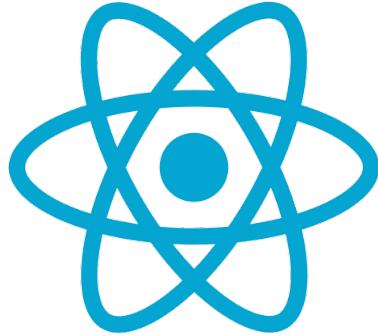
Creata da Facebook, React è la colonna portante del social network più popolare del mondo e su di essa si basa l'interfaccia Web di Instagram.

La creazione di applicazioni Web, indipendentemente dal framework scelto per lo sviluppo, coinvolge necessariamente i tre linguaggi fondamentali della piattaforma: HTML per la struttura, CSS per la stilizzazione e JavaScript per la logica applicativa.

Le pagine complete invece sono ridotte al minimo, anzi molto spesso a una sola, tant'è vero che queste applicazioni prendono il nome di Single Page Applications (SPA) e che servono da "contenitore" in cui creare e gestire l'interfaccia utente.

La forza di React rispetto ad altre librerie è quella di consentire l'uso di un approccio dichiarativo simile all'HTML, quindi molto familiare, per definire i componenti che rappresentano parti significative e logiche dell'interfaccia utente, ad esempio un commento a un articolo, o la lista degli stessi commenti.

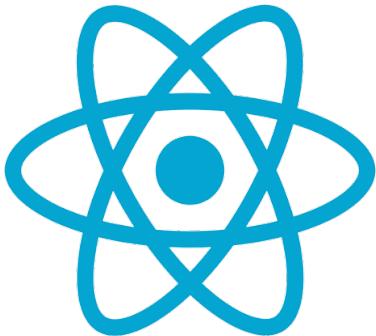
---



# Perché usare REACTJS

- È open source, vale a dire che è possibile contribuire a esso inviando problemi o richieste pull. (*Proprio come questi documenti*)
- È dichiarativo, ovvero si scrive il codice desiderato e React accetta il codice dichiarato ed esegue tutti i passaggi JavaScript/DOM per ottenere il risultato desiderato.
- Si tratta di un componente, vale a dire che le applicazioni vengono create usando moduli di codice indipendenti prefabbricati e riutilizzabili che gestiscono il proprio stato e possono essere associati con il framework di React, rendendo possibile passare i dati attraverso l'app mantenendo lo stato fuori dal DOM.
- Il motto React è "Imparare una volta, scrivere ovunque". L'intenzione è quella di riutilizzare il codice e di non fare ipotesi su come si userà React utente con altre tecnologie, ma per rendere riutilizzabili i componenti senza la necessità di riscrivere il codice esistente.

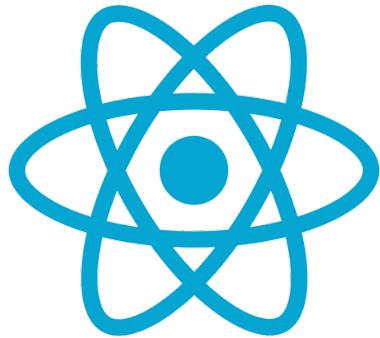
# Perché usare REACTJS



- JSX è un'estensione di sintassi per JavaScript scritta da usare con React simile a HTML, ma è in realtà un file JavaScript che deve essere compilato o convertito in JavaScript normale.
- DOM virtuale: DOM è l'acronimo di Document Object Model e rappresenta l'interfaccia utente dell'app. Ogni volta che lo stato dell'interfaccia utente dell'app cambia, il DOM viene aggiornato per rappresentare la modifica. Quando un DOM viene aggiornato di frequente, le prestazioni diventano lente. Un DOM virtuale è solo una rappresentazione visiva del DOM, quindi quando lo stato dell'app cambia, il DOM virtuale viene aggiornato anziché il DOM reale, riducendo il costo delle prestazioni. Si tratta di un'appresentazione di un oggetto DOM, ad esempio una copia leggera.
- Visualizzazioni sono gli elementi visualizzati dall'utente visualizzati nel browser. In React, la visualizzazione è correlata al concetto di elementi di rendering che si desidera che un utente visualizzi sullo schermo.

---

# Perché usare REACTJS



- Stato: fa riferimento ai dati archiviati da visualizzazioni diverse. Lo stato si basa in genere sull'utente e sulle operazioni eseguite dall'utente. Ad esempio, l'accesso a un sito Web può mostrare il profilo utente (visualizzazione) con il nome (stato). I dati sullo stato cambieranno in base all'utente, ma la visualizzazione rimarrà invariata.

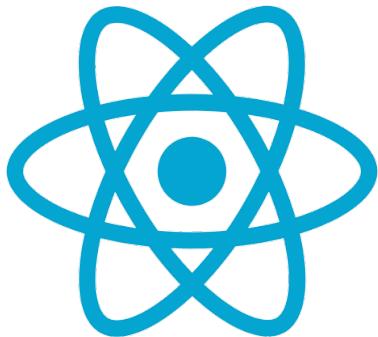
Benché dichiarativa, la rappresentazione del componente in realtà si traduce in chiamate all'API di React che intervengono - nel modo più veloce e performante possibile - sul DOM della pagina per creare gli elementi necessari.

---

---

# Differenze con JQuery

Rispetto a JQuery, React non richiede l'assegnazione obbligatoria di ID univoci o classi, né richiede allo sviluppatore di intervenire direttamente sul DOM, ma è la libreria stessa che si fa carico di questo compito in base alla struttura del componente dichiarato in primis e, in secondo luogo, in base allo stato interno del componente Web e alla variazione dei valori delle proprietà assegnate allo stesso.



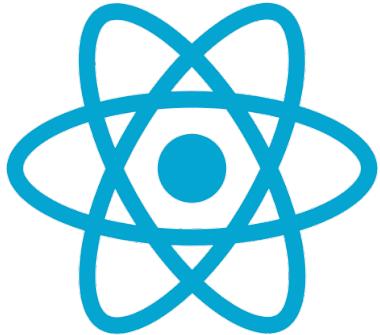
In breve, React esclude un intervento diretto sul DOM, lasciando a noi il compito di definire i componenti che costituiscono l'interfaccia dell'applicazione e i dati da utilizzare per la generazione del markup. Sarà la libreria ad intervenire sul DOM di conseguenza, utilizzando peraltro il modo che garantisce le maggiori performance possibili.

---

---

# Differenze con AngularJS

AngularJS è una soluzione più completa, a discapito di una complessità maggiore di apprendimento e uso, laddove React si presenta più focalizzato e accessibile all'apprendimento, con la possibilità di ricorrere a librerie note (es. anche la stessa JQuery) per le parti che non sono strettamente legate all'interfaccia utente.



La contestazione maggiore che viene mossa a React da parte degli sviluppatori AngularJS è quella di mescolare logica di gestione degli eventi alla presentazione, riducendo quindi la separazione delle responsabilità degli elementi del progetto.

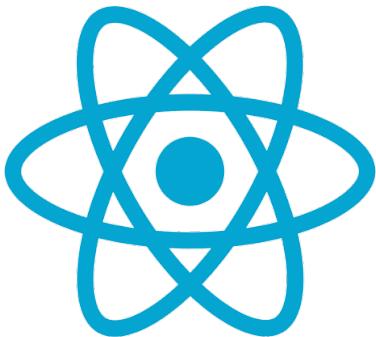
Questo approccio viene giustificato da React con il criterio della “Separation of Concerns” a discapito della “Separation of Responsibility”: in termini pratici, con React si tende a isolare codice e lo stato dei componenti in base al loro ambito di utilizzo, rendendoli il più possibile autonomi e riutilizzabili in un medesimo contesto, piuttosto che estremizzare la separazione della parte di presentazione e del codice che la governa.

---

# Usiamo la libreria react e ReactDOM

React è una libreria e possiamo importarlo ed usarlo in qualsiasi pagina html per aiutarci a generare layout reattivi.

## React



React è una libreria JavaScript sviluppata da Facebook per costruire interfacce utente. È basata sulla creazione di componenti riutilizzabili, che gestiscono il proprio stato e si compongono per formare applicazioni complesse. React è focalizzata principalmente sulla creazione della vista dell'applicazione, permettendo agli sviluppatori di definire come l'app dovrebbe apparire in base ai cambiamenti nei dati.

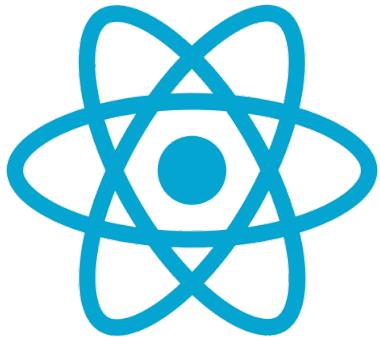
## ReactDOM

ReactDOM è un pacchetto che fornisce metodi specifici del DOM (Document Object Model) che possono essere utilizzati con React. La funzione più nota di ReactDOM è `ReactDOM.render()`, che viene utilizzata per montare un componente React al DOM. Questo è il punto di ingresso per un'applicazione React in una pagina web, collegando la logica React all'interfaccia utente definita nel markup HTML.

```
<!-- React e ReactDOM --><script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script> <script crossorigin
src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
```

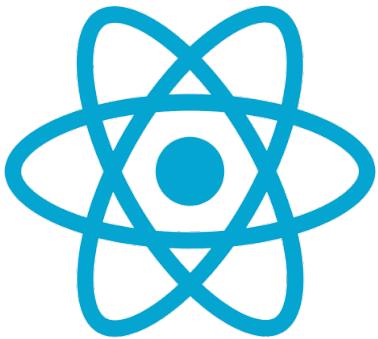
# Usiamo la libreria react e ReactDom

React è una libreria e possiamo importarlo ed usarlo in qualsiasi pagina html per aiutarci a generare layout reattivi. Creiamo una semplice pagina HTML ed importiamo i file react.js e reactdom.js



```
2  <html>
3
4      <head>
5          <meta charset="UTF-8" />
6          <title>Hello World</title>
7          <!--
8              <script src="https://unpkg.com/react@18/umd/react.development.js"><
9                  <script src="https://unpkg.com/react-dom@18/umd/react-dom.developme
10             -->
11             <script src="react.js"></>
12             <script src="reactdom.js"></>
13         </head>
14
15         <body>
16             <div id="root"></div>
17
18
19         </body>
20
21     </html>
```

# Usiamo la libreria react e ReactDOM



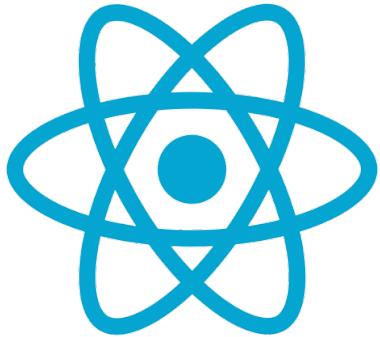
Nel div andremo a renderizzare i nostri componenti. Andiamo a creare subito un elemento di react. L'app deve avere un componente principale che sarà root

```
<body>
  <div id="root"></div>

  <script>
    const rootElement=document.getElementById("root");
    const root=ReactDOM.createRoot(rootElement);
    console.log(rootElement);
  </script>
</body>
```

---

# Usiamo la libreria react e ReactDOM



Consideriamo la creazione di un componente `<h1>` in una normale pagina HTML.

Inizialmente, questa pagina è statica, senza interattività. Si potrebbe pensare di aggiungere interattività manualmente, utilizzando event listener direttamente sul DOM. Tuttavia, React opera in modo diverso: non interagisce direttamente con il DOM della pagina. Al contrario, React gestisce un Virtual DOM - una rappresentazione leggera in memoria.

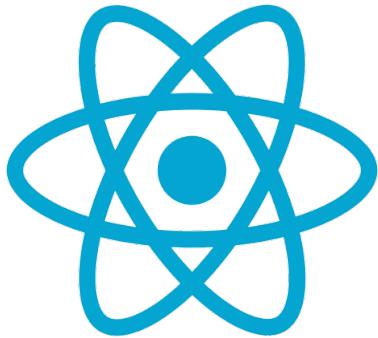
Quando si aggiunge interattività in React, ad esempio aggiungendo un listener di eventi a un componente, React aggiorna il Virtual DOM. Dopodiché, React confronta il Virtual DOM aggiornato con il DOM reale tramite un processo chiamato '**reconciliation**'. Basandosi sulle differenze rilevate, React applica in modo efficiente le modifiche necessarie al DOM reale.

Questo approccio è più performante rispetto all'interazione diretta con il DOM e permette a React di gestire le proprietà degli elementi e lo stato dell'applicazione in modo più efficace.

---

# Usiamo la libreria react e ReactDOM

Possiamo creare elementi HTML e passargli gli attributi, tipo style, class che diventa className ed eventi tipo onClick.



```
<script>
    const rootElement = document.getElementById('root');

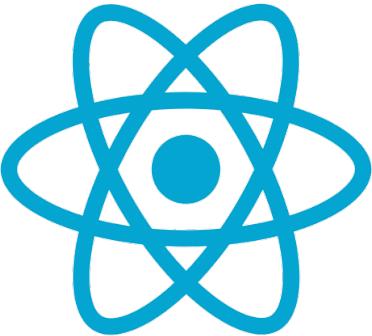
    const root = ReactDOM.createRoot(rootElement);

    const h1 = React.createElement('h1', {
        style: {
            color: 'green',
            width: '200px'
        },
        className: 'big',
        onClick: ele => { console.log(ele.target) }
    },
    'Hello World'

);
root.render(h1);
</script>
```

# Usiamo la libreria react e ReactDOM

Supponiamo ora di voler creare un div chiamiamolo `main` e renderizziamo al suo interno `h1`.



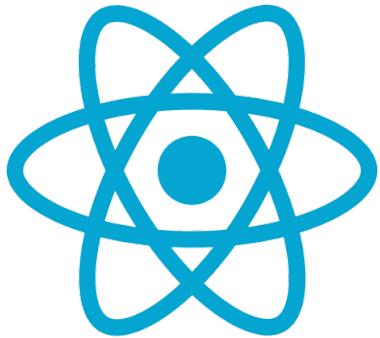
```
        onClick: ele => { console.log(ele.target) }
    },
    'Hello World'

);
const div = React.createElement('div', {
    className: 'main'
},
h1
);
```

```
<style>
    .main {
        background-color: brown;
        color: white;
        border: 5px;
        border-radius: 10px;
    }
</style>
```

# Usiamo la libreria react e ReactDom

Vediamo ora con aggiungere un altro elemento come una lista

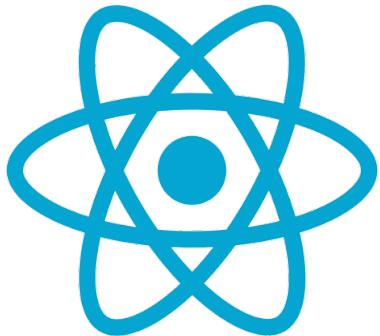


```
const ul=React.createElement("ul",{
  className:"list"
},[
  React.createElement("li",{key:'php'},'PHP'),
  React.createElement("li",{key:'javascript'},'Javascript'),
  React.createElement("li",{key:'angular'},'Angular')
])
```

---

# Usiamo la libreria react e ReactDOM

Nel contesto di React, la prop `key` è utilizzata per aiutare React a identificare quali elementi sono cambiati, sono stati aggiunti, o sono stati rimossi tra diversi render. Le keys dovranno essere assegnate agli elementi all'interno di un array per dare agli elementi una identità stabile:



Quando si crea un elenco di elementi React con `React.createElement`, l'uso della prop `key` è particolarmente importante in situazioni dove l'ordine degli elementi può cambiare tra i render. Senza una key che identifichi ciascun elemento in maniera univoca, React non avrà modo di sapere quale elemento è stato modificato, aggiunto o rimosso, il che potrebbe portare a prestazioni inefficienti e potenziali errori nello stato dell'interfaccia utente.

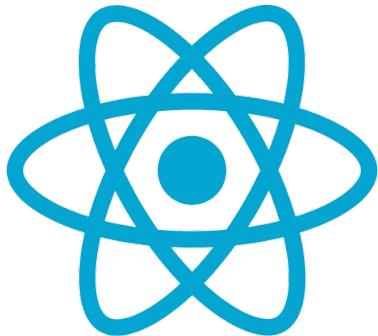
Le keys 'php', 'javascript' e 'angular' sono utilizzate per identificare univocamente ciascun `<li>` nell'array. Questo significa che se l'ordine degli elementi cambia in un futuro render, React sarà in grado di riconoscere quali elementi sono stati modificati basandosi sulla loro key e sarà in grado di evitare un rirender completo dell'intera lista. Questo è fondamentale per ottimizzare le prestazioni quando si lavora con liste dinamiche che possono subire mutazioni nel tempo.

---

# Usiamo la libreria react e ReactDOM

---

Ecco un esempio di come React potrebbe utilizzare le keys per aggiornare l'interfaccia utente:



- Se l'elemento con key 'php' viene rimosso, React eliminerà solo quell'elemento dalla lista, invece di dover rirenderizzare l'intera lista.
- Se un elemento con key 'sql' viene aggiunto all'inizio dell'array, React aggiungerà quell'elemento all'inizio della lista nel DOM senza rirenderizzare gli altri elementi già presenti.

In assenza di keys, React non avrebbe altra scelta se non quella di rirenderizzare tutti gli elementi dell'array per garantire che l'interfaccia utente rifletta l'effettiva sequenza degli elementi nell'array, perché non sarebbe in grado di identificare quale elemento specifico è cambiato.

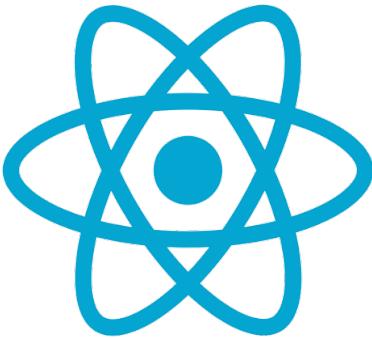
---

# Usiamo babel e jsx

Creare elementi in questo modo non è certo molto comodo, per questo react ci da a disposizione un linguaggio che si chiama jsx che ci permette di scrivere javascript come fosse html. Questo è possibile tramite la libreria babel che interpreta il jsx.

Andiamo quindi a creare una nuova pagina e importiamo sia i file di react che la libreria babel. Importiamo anche un file con estensione jsx.

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```



```
<!DOCTYPE html>
<html lang="it">

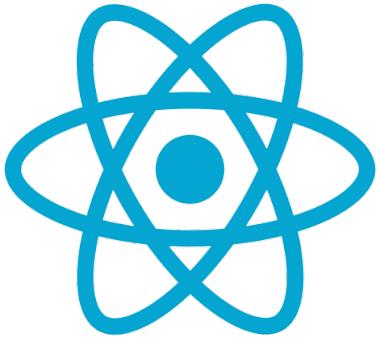
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>React with babel</title>
  </head>

  <body>
    <div id="root"></div>
    <!-- end of the page -->
    <script src="https://unpkg.com/react@18/umd/react.prod.js" crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.prod.js" crossorigin></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
    <script src="script.jsx" type="text/babel"></script>
  </body>

</html>
```

# Usiamo babel e jsx

Nel file jsx creiamo la root della nostra app e poi nella funzione render andiamo a scrivere il nostro HTML



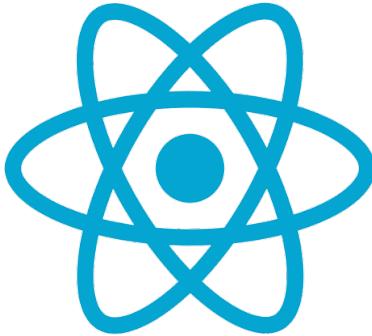
```
const rootEle = document.getElementById('root');

const root = ReactDOM.createRoot(rootEle);

root.render(
  <div>
    <h1>Using JSX</h1>
    <ul>
      <li>JAVASCRIPT</li>
      <li>PHP</li>
      <li>HTML</li>
    </ul>
  </div>
);
```

# Usiamo babel e jsx

Strutturiamo ora elementi creati da noi tramite funzioni.



```
const root = ReactDOM.createRoot(rootEle);
const App = (props) => {
  return (
    <main className='main'>
      <h1>This is the main app</h1>
      {props.children}
    </main>
  );
};
root.render(
  <>
    <App>
      <h1>Using JSX</h1>
      <ul>
        <li>JAVASCRIPT</li>
        <li>PHP</li>
        <li>HTML</li>
      </ul>
    </App>
  </>
);
```

```
const rootEle = document.getElementById('root');

const root = ReactDOM.createRoot(rootEle);

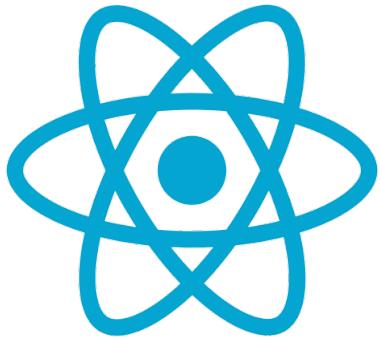
const App = ({ children }) => {
  return (
    <main className='main'>
      <h1>This is the main app</h1>
      {children}
    </main>
  );
};

function List() {
  return (
    <ul>
      <li>JAVASCRIPT</li>
      <li>PHP</li>
      <li>HTML</li>
    </ul>
  );
}

root.render(
  <>
    <App>
      <h2>My learning path</h2>
      <List></List>
    </App>
  </>
);
```

# Cos'è NODE.JS

Node.js è una runtime di JavaScript Open source multiplataforma orientato agli eventi per l'esecuzione di codice JavaScript, costruita sul motore JavaScript V8 di Google Chrome.

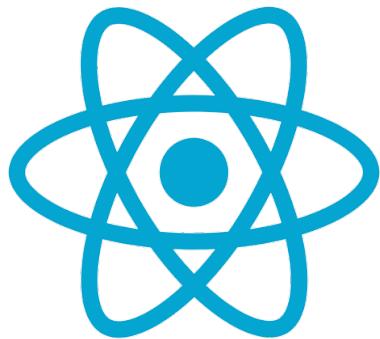


In origine JavaScript veniva utilizzato principalmente lato client. In questo scenario gli script JavaScript, generalmente incorporati all'interno dell'HTML di una pagina web, vengono interpretati da un motore di esecuzione incorporato direttamente all'interno di un Browser. Node.js consente invece di utilizzare JavaScript anche per scrivere codice da eseguire lato server, ad esempio per la produzione del contenuto delle pagine web dinamiche prima che la pagina venga inviata al Browser dell'utente.

Node.js come detto è un runtime di javascript ovvero un programma che, grazie al «motore» V8 di Google Chrome, viene eseguito codice JavaScript lato client

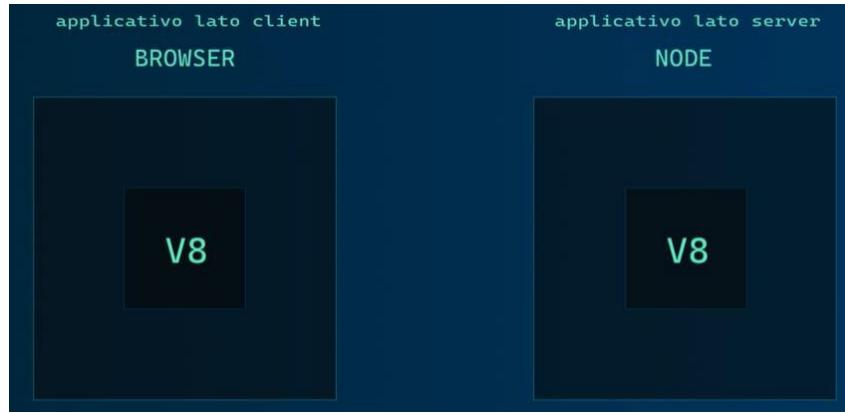
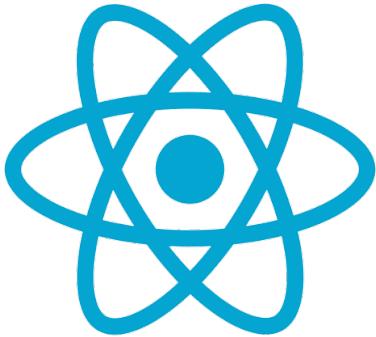
Quando parliamo di «motore» JavaScript parliamo di quel componente sviluppato inizialmente solo per il browser, il cui compito è quello di prendere il nostro codice sorgente JavaScript e trasformarlo in codice macchina:

# Cos'è NODE.JS



Ogni browser ha il suo «motore» JavaScript specifico Ad esempio FireFox ha « SpiderMonkey », mentre chrome ha «V 8 » Inizialmente JavaScript è nato come linguaggio lato client e quindi eseguito dal «motore» residente all'interno del browser.

# Cos'è NODE.JS

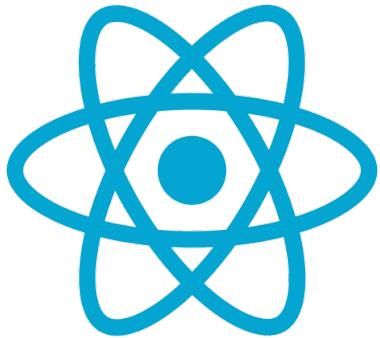


Nel 2009 l'idea «rivoluzionaria» del progettista di Node.js Ryan Dahl è stata quella di prendere il motore V8 ed inserirlo all'interno di un nuovo ambiente in cui eseguire codice JavaScript. Questo ambiente prende il nome di Node.js che non opera lato client ma opera lato server

---

# Cos'è NODE.JS

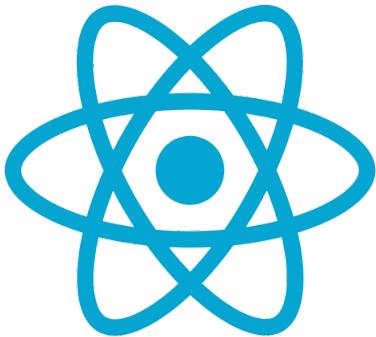
Abbiamo quindi 1 motore JavaScript V 8 e due runtime diversi, uno lato client ed uno lato server Entrambi questi runtime espandono le funzionalità di V 8 in base alla propria esigenza



Ad esempio l'oggetto `window` e l'oggetto `document` presente nel runtime browser non è presente nel runtime nodejs in quanto non abbiamo lato server una finestra del browser da gestire, ma ha un altro oggetto necessario al server che vedremo più avanti, cioè l'oggetto **global**

# Cos'è NPM

NPM (Node Package Manager) è il gestore di pacchetti ufficiale di Node.js. È uno strumento fondamentale per lo sviluppo JavaScript moderno.



## Cosa fa NPM?

Installa librerie - Scarica e gestisce dipendenze

Gestisce versioni- Controlla compatibilità tra pacchetti

Esegue script- Comandi personalizzati per il progetto

Pubblica pacchetti - Condivide codice con la community

## Componenti principali

### 1. Registry NPM

- Database online con milioni di pacchetti JavaScript
- Accessibile su [npmjs.com](https://npmjs.com)
- Pacchetti gratuiti e a pagamento

### 2. CLI (Command Line Interface)

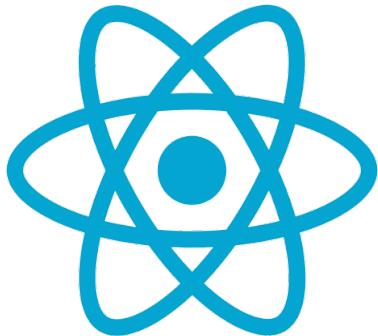
- Strumento da terminale per interagire con NPM
- Si installa automaticamente con Node.js
- Comandi come `npm install`, `npm start`, ecc.

### 3. package.json

- File di configurazione del progetto
- Elenca dipendenze, script, metadati
- "Carta d'identità" del progetto

# Approccio Asincrono

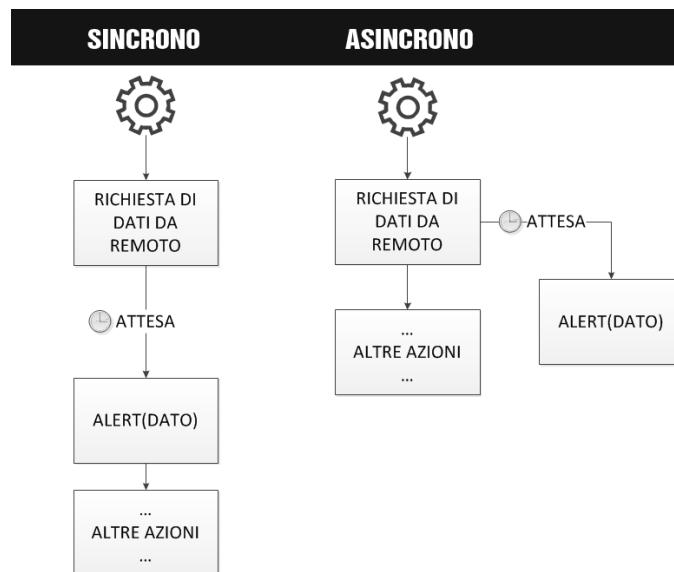
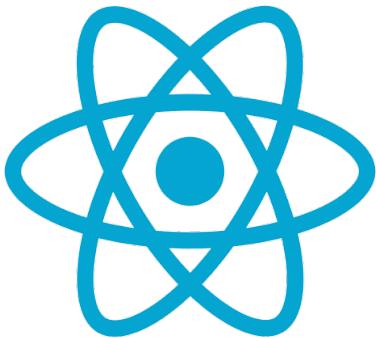
La caratteristica principale di Node.js risiede nella possibilità che offre di accedere alle risorse del sistema operativo in modalità **event-driven** e non sfruttando il classico modello basato su processi o thread concorrenti, utilizzato dai classici web server.



Il modello event driven o “programmazione ad eventi”, si basa su un concetto piuttosto semplice, si lancia una azione quando accade qualcosa Ogni azione quindi risulta asincrona a differenza dei pattern di programmazione più comune in cui un’azione succede ad un’altra solo dopo che essa è stata completata

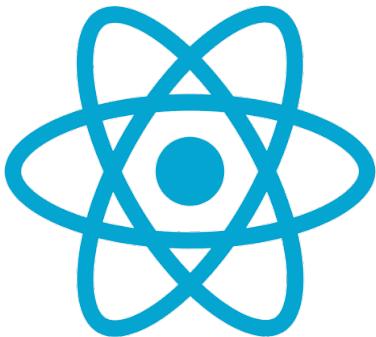
# Approccio Asincrono

Grazie al comportamento asincrono durante le attese di una certa azione il runtime può gestire qualcos'altro che ha a che fare con la logica applicativa, ad esempio



# Installazione Node.js

Per installare localmente Node.js bisogna andare sul sito ufficiale [www.nodejs.org](http://www.nodejs.org) e scaricare il pacchetto per il nostro sistema operativo



Node.js® è un runtime JavaScript costruito sul motore JavaScript V8 di Chrome.

Join us at OpenJS World, a free virtual event on June 2-3, 2021

Download per Windows (x64)

14.17.0 LTS

Consigliata

16.2.0 Corrente

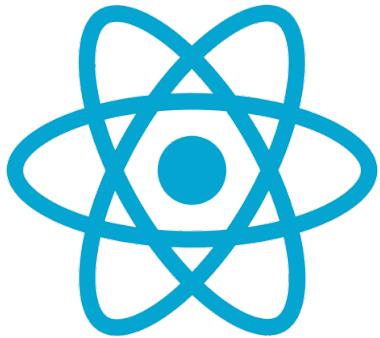
Ultime funzionalità

[Altri download](#) | [Changelog](#) | [Documentazione API](#)    [Altri download](#) | [Changelog](#) | [Documentazione API](#)

Dai un'occhiata alla tabella di marcia LTS.

# Installazione Node.js

Node.js va utilizzato da riga di comando. Verifichiamo che l'installazione è andata a buon fine aprendo il terminale dos e digitare il comando node v



```
ie utente suo nome
ca Amministratore: Prompt dei comandi
Microsoft Windows [Versione 10.0.19042.985]
(c) Microsoft Corporation. Tutti i diritti sono riservati.

C:\WINDOWS\system32>node -v
v14.17.0

C:\WINDOWS\system32>
```

---

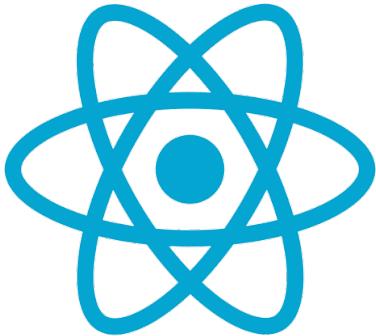
# Il nostro primo script Node

Per prima cosa apriamo il nostro terminale e digitiamo semplicemente il comando node In questo modo entriamo in una modalità interattiva per poter fare dei semplici test.

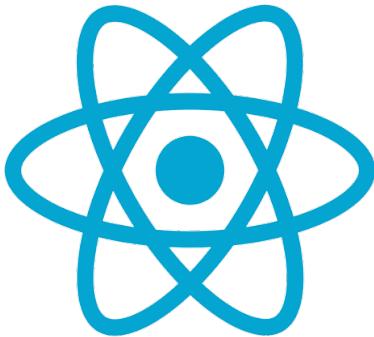
Proviamo a scrivere la seguente riga di codice:

```
'node.js'.toUpperCase()
```

e poi diamo invio Possiamo vedere il risultato a video. Per uscire dalla modalità node premiamo i tasti CTRL+D



# Il nostro primo script Node

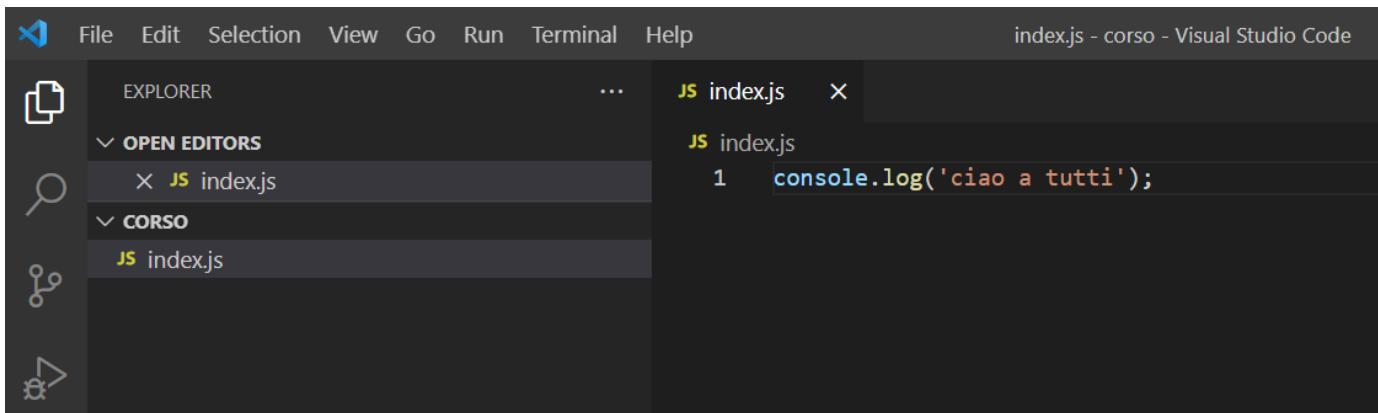
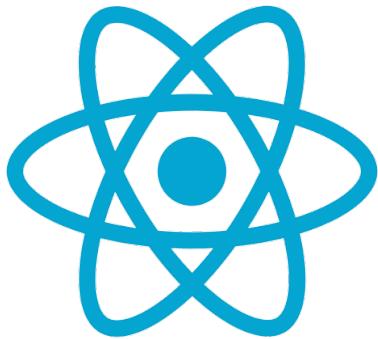


```
C:\WINDOWS\system32>node
Welcome to Node.js v14.17.0.
Type ".help" for more information.
> 'node.js'.toUpperCase()
'NODE.JS'
>

C:\WINDOWS\system32>
```

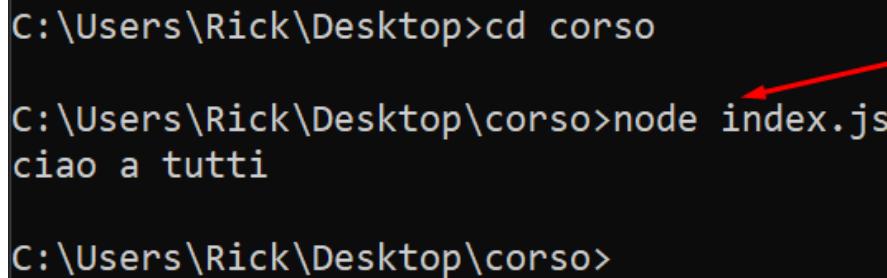
Per fare un esempio meno banale andiamo a creare una cartella sul desktop e la chiamiamo «corso» e sempre tramite la riga di comando ci andremo a posizionare all'interno di essa e creeremo un file chiamato index.js (usando un editor di testo come ad esempio Visual Studio Code)

# Il nostro primo script Node



A screenshot of the Visual Studio Code interface. The title bar reads "index.js - corso - Visual Studio Code". The left sidebar shows the "EXPLORER" view with "OPEN EDITORS" expanded, showing "index.js" under both "File Explorer" and "Corso". The main editor area shows the "index.js" file with the following content:

```
JS index.js
1 console.log('ciao a tutti');
```



A screenshot of a terminal window with a black background and white text. The command "cd corso" is entered, followed by "node index.js", which outputs "ciao a tutti". A red arrow points from the word "ciao" in the terminal output towards the "ciao a tutti" line in the VS Code editor.

```
C:\Users\Rick\Desktop>cd corso

C:\Users\Rick\Desktop\corso>node index.js
ciao a tutti

C:\Users\Rick\Desktop\corso>
```

# Il registro NPM

NPM rappresenta un repository di librerie scritte apposta per poter essere utilizzate con Node.js

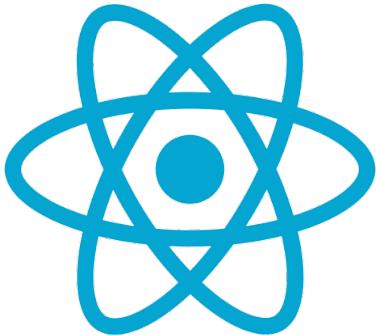
Per installare un pacchetto basta un semplice:

```
npm install nome pacchetto
```

Per vedere un esempio concreto consideriamo l'installazione del modulo socket.io Questo modulo permette di gestire le connessioni via socket a basso livello.

Non preoccupiamoci di capire in dettaglio a cosa serve al momento lo useremo solo come esempio di procedura di installazione di un nuovo componente

```
npm install bootstrap
```

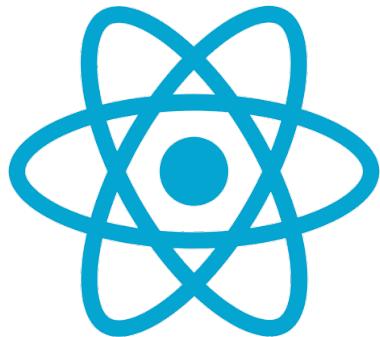


---

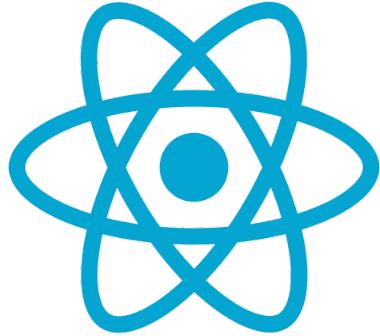
# Come creare una React App

Per creare la prima app è sufficiente digitare il comando npx che non è altro che un esecutore di pacchetti seguito dal pacchetto create-react-app e poi il nome dell'applicazione Per fare un

primissimo esempio creiamo sul desktop una cartella ciaomondo e tramite terminale  
digitiamo:



```
npx create-react-app prima-app
```



# Come creare una React App

```
Success! Created ciao at C:\Users\Rick\Desktop\ciao
Inside that directory, you can run several commands:
```

```
npm start
  Starts the development server.
```

```
npm run build
  Bundles the app into static files for production.
```

```
npm test
  Starts the test runner.
```

```
npm run eject
  Removes this tool and copies build dependencies, configuration files
  and scripts into the app directory. If you do this, you can't go back!
```

We suggest that you begin by typing:

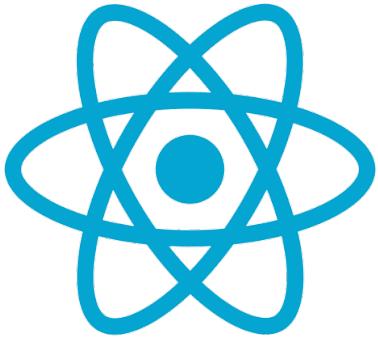
```
cd ciao
npm start
```

Happy hacking!

```
C:\Users\Rick\Desktop>
```

# Come creare una React App

Appena lanciamo il comando partirà la configurazione della nostra nuova app, ci vuole qualche secondo Una volta terminata l'installazione di tutte le librerie necessarie possiamo lanciare il nostro server locale digitando da riga di comando



`npm start`

```
Compiled successfully!

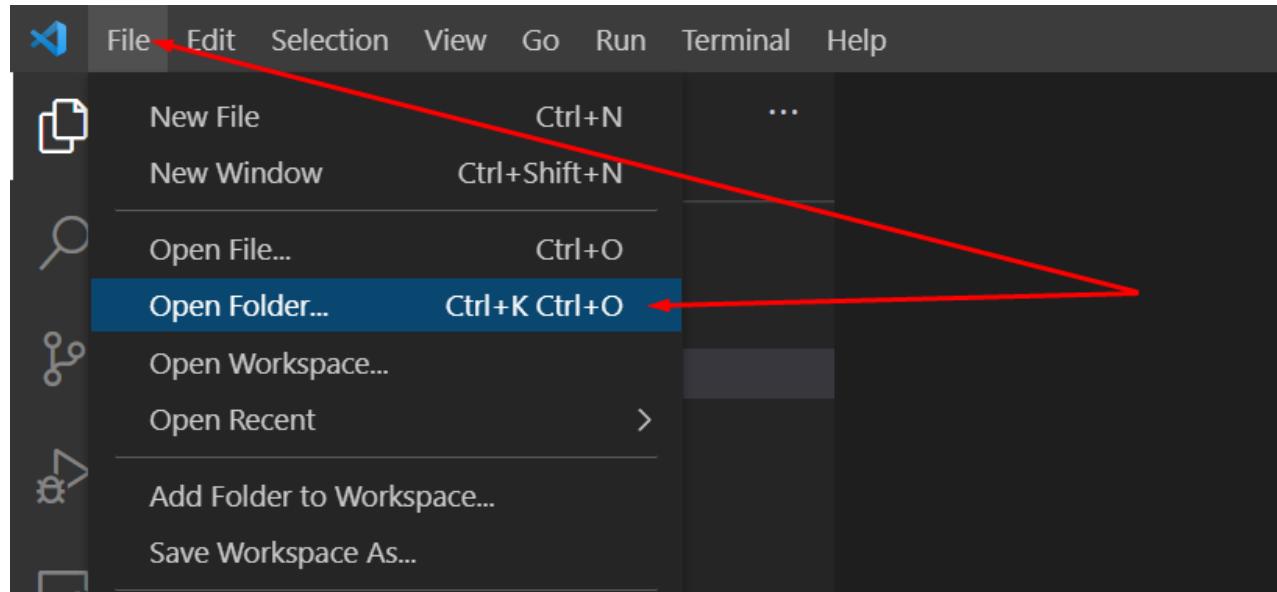
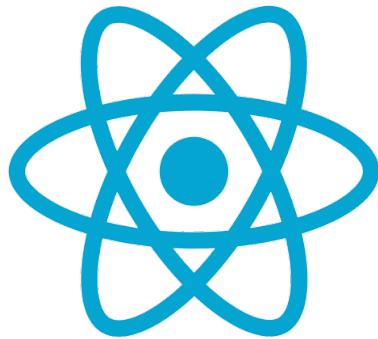
You can now view ciao in the browser.

  Local:          http://localhost:3000
  On Your Network: http://172.18.144.1:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

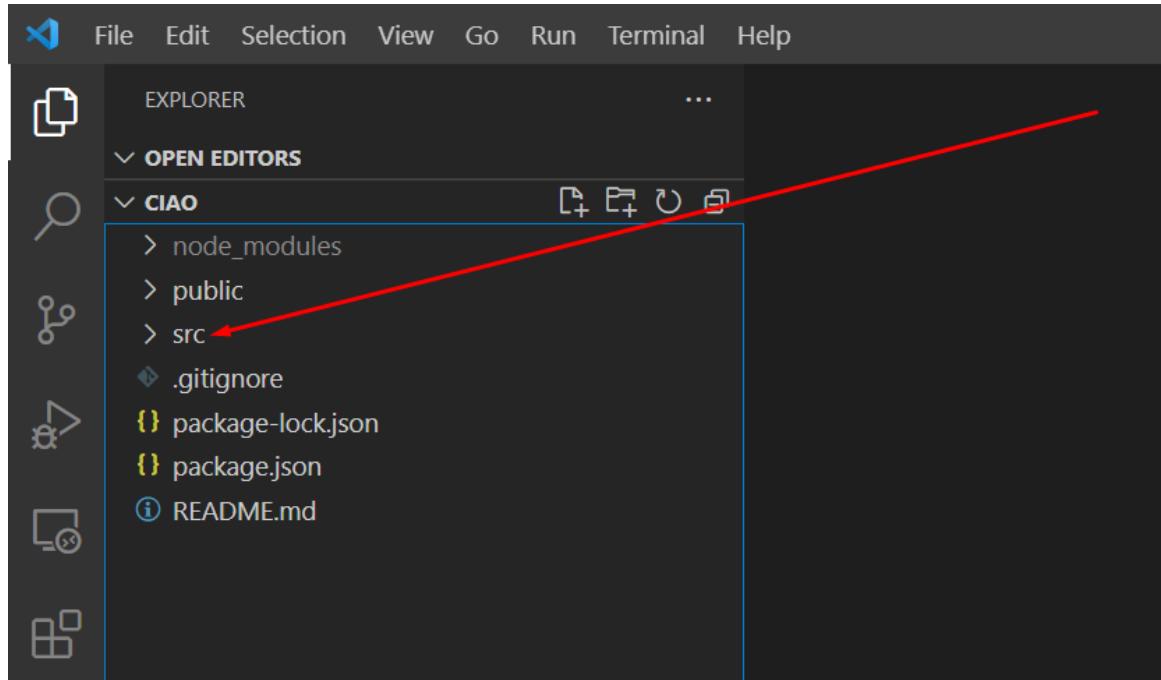
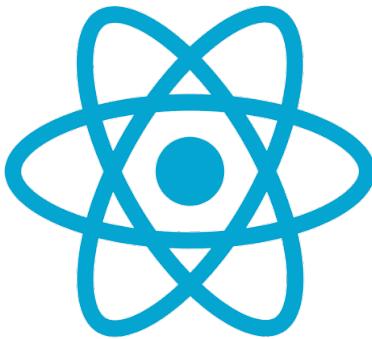
# Struttura di un progetto React

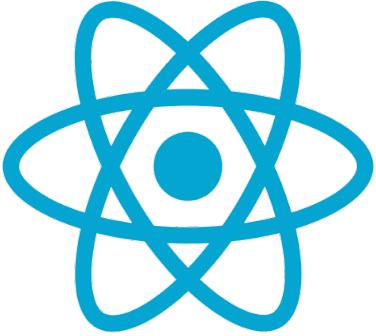
Andiamo ora ad aprire la nostra applicazione attraverso Visual Studio Code



# Struttura di un progetto React

Andiamo ora ad aprire la nostra applicazione attraverso Visual Studio Code





# Struttura di un progetto React

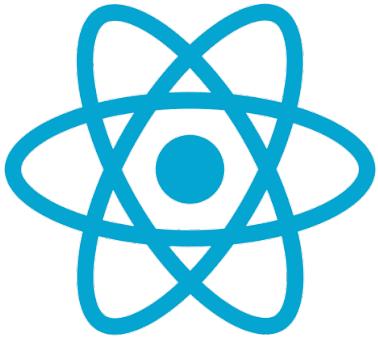
Per fare una prima prova apriamo il file App.js che troviamo all'interno della nostra cartella src e proviamo a modificare il testo all'interno del codice html ad esempio sostituiamo la scritta «Learn React» con «Ciao Mondo», salviamo e vediamo cosa succede (per vederlo affianchiamo le due finestre browser e visual studio code)

The image shows a dual-monitor setup. On the left monitor, the Visual Studio Code interface is displayed. The Explorer sidebar shows a project structure with files like App.css, App.test.js, index.css, index.js, logo.svg, reportWebVitals.js, setupTests.js, .gitignore, package-lock.json, package.json, and README.md. The App.js file is open in the editor, showing the following code:

```
File Edit Selection View Go Run ... App.js - ciao - Visual Studio Code ...
OPEN EDITORS
JS App.js src M
src > App.js > App
1 import logo from './logo.svg';
2 import './App.css';
3
4 function App() {
5   return (
6     <div className="App">
7       <header className="App-header">
8         <img src={logo} className="App-logo" alt="React logo" />
9         <p>
10           Edit <code>src/App.js</code>
11           <a href="https://reactjs.org" target="_blank" rel="noopener noreferrer">
12             Ciao Mondo
13           </a>
14         </p>
15       </header>
16     </div>
17   );
18 }
19
20 export default App;
```

The right monitor displays a web browser window showing the application running at localhost:3000. The page contains a header with the React logo and a paragraph with the text "Ciao Mondo". A red arrow points from the word "Ciao Mondo" in the browser back to the corresponding line in the App.js code in VS Code. Another red arrow points from the text "Edit src/App.js and save to reload." in the browser back to the "Edit" link in the code editor. The browser's address bar shows "localhost:3000".

# Struttura di un progetto React



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help
- Explorer:** Shows the project structure:
  - OPEN EDITORS: App.js, package.json
  - CIAO: node\_modules, public, src
    - src: App.css, App.js, App.test.js, index.css, index.js, logo.svg, reportWebVitals.js, setupTests.js
    - package-lock.json
    - package.json
    - README.md
  - .gitignore
- Code Editor:** package.json (highlighted)
- Terminal:** package.json - ciao - Visual Studio Code

A red arrow points from the title "Struttura di un progetto React" down to the package.json file in the code editor.

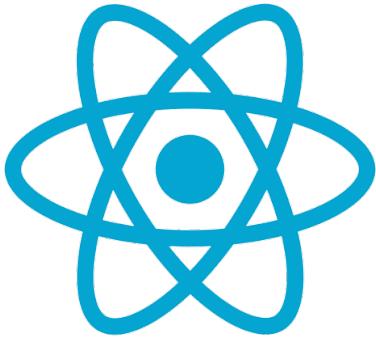
```
1  {
2   "name": "ciao",
3   "version": "0.1.0",
4   "private": true,
5   "dependencies": {
6     "@testing-library/jest-dom": "^5.13.0",
7     "@testing-library/react": "^11.2.7",
8     "@testing-library/user-event": "^12.8.3",
9     "react": "^17.0.2",
10    "react-dom": "^17.0.2",
11    "react-scripts": "4.0.3",
12    "web-vitals": "^1.1.2"
13  },
14  "scripts": {
15    "start": "react-scripts start",
16    "build": "react-scripts build",
17    "test": "react-scripts test",
18    "eject": "react-scripts eject"
19  },
20  "eslintConfig": {
21    "extends": [
22      "react-app",
23      "react-app/jest"
24    ]
25  },
26}
```

---

# Struttura di un progetto React

Questo è il primo file del nostro progetto che dobbiamo conoscere package.json dove troviamo la nostra configurazione e tutte le librerie che sono state installate automaticamente quando abbiamo creato il progetto

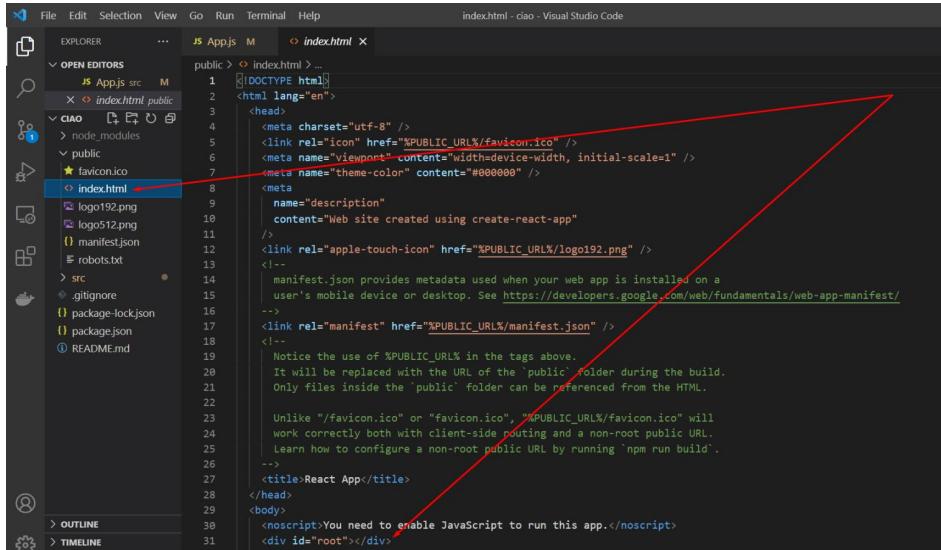
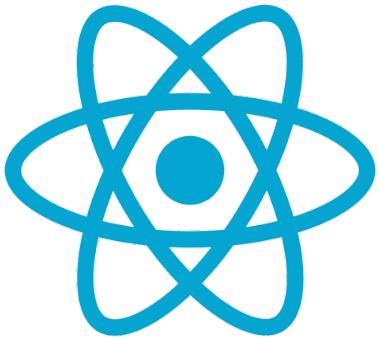
Fisicamente tutte le librerie scaricate possiamo vederle all'interno della cartella node\_modules



# Struttura di un progetto React

All'interno della cartella public troviamo il file index.html che è il file che effettivamente viene caricato quando facciamo partire la nostra applicazione.

Tutto quello che scriveremo all'interno del nostro progetto, quindi nella cartella src verrà visualizzato all'interno del div con id root.



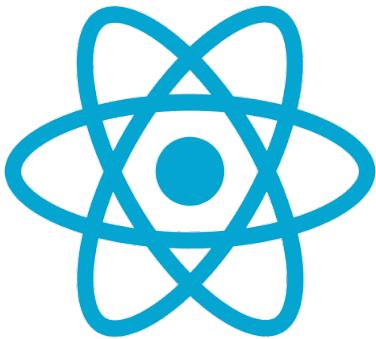
The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure:
  - JS App.js
  - src
  - index.html (highlighted with a red arrow)
  - public
  - node\_modules
  - favicon.ico
  - manifest.json
  - robots.txt
  - src
  - .gitignore
  - package-lock.json
  - package.json
  - README.md
- Editor:** The index.html file is open, showing its content. A red arrow points from the word "index.html" in the file path in the Explorer to the opening tag in the editor.

```
public > index.html > ...
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app" />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the 'public' folder during the build.
      Only files inside the 'public' folder can be referenced from the HTML.
    -->
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

---

# Esercizio

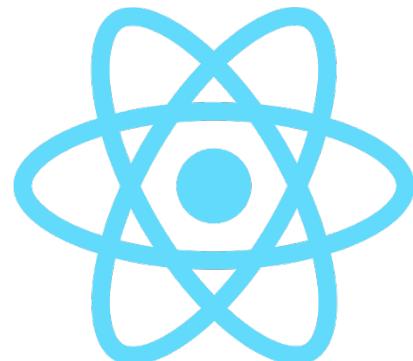


Creare un progetto react e creare una pagina di presentazione personalizzata sostituendo l'html presente di default nella pagina App.js

Importare bootstrap o con il semplice link html o scaricando la libreria con npm

# React

Le basi



# Componenti Funzionali

Dopo aver visto la struttura di un progetto andiamo su App.js vediamo come poter scrivere codice JavaScript.

In App.js possiamo vedere come si dichiara un componente in React, componente funzionale. I componenti funzionali sono stati introdotti in React per la prima volta con la versione 0.14 nel 2015. Tuttavia, l'uso di componenti funzionali è diventato molto più prevalente con l'introduzione degli Hooks in React nella versione 16.8, rilasciata a febbraio 2019. Gli Hooks hanno permesso ai componenti funzionali di utilizzare funzionalità che prima erano possibili solo nei componenti basati su classi, come lo stato interno e gli effetti collaterali (use Effect), rendendoli molto più potenti e flessibili.

Prima dell'introduzione degli Hooks, i componenti funzionali erano comunemente chiamati "stateless functional components" (SFC) o "dumb components" perché erano principalmente usati per ricevere props e restituire JSX senza gestire lo stato o altri aspetti del ciclo di vita del componente. Con gli Hooks, questa limitazione è stata superata, permettendo agli sviluppatori di utilizzare pienamente i componenti funzionali per quasi tutti gli scenari di sviluppo, portando a una crescente preferenza per questa sintassi più concisa e moderna rispetto ai classici componenti basati su classi.

# JSX

Abbiamo detto che JSX è JavaScript scritto simil HTML e interpretato da Babel.  
Se vogliamo scrivere codice JS al suo interno dobbiamo farlo tra le parentesi {}.  
Puliamo tutto App.js e proviamo a scrivere un po' di JS.  
Prima però alcune indicazioni su cosa si può usare e cosa no.

**SI - Espressioni JavaScript:** Puoi incorporare qualsiasi espressione JavaScript tra parentesi graffe {}. Questo include variabili, funzioni, operazioni matematiche, ecc.

```
function App() {  
  const name = "Mondo";  
  return <div>Ciao, {name}!</div>;
```

**SI - Operatore Ternario e Logica booleana:**  
Per le condizioni, spesso si utilizzano operatori ternari o la logica booleana.

```
function App() {  
  return <div>{isLoggedIn ? <UserPanel /> : <LoginButton />}</div>;
```

**SI - Componenti:** Puoi usare componenti come tag, sia componenti funzionali che classi. Questi possono essere inclusi e annidati proprio come gli elementi HTML.

```
function App() {  
  return <MyCustomComponent />;
```

# JSX

**SI – Metodi iterativi Array:** Il metodo più comune per ciclare attraverso array in JSX è l'uso del metodo `.map()`, che è perfetto per trasformare gli array in elementi di React.

```
function App() {  
  return (  
    <ul>  
      {items.map((item, index) => (  
        <li key={index}>{item}</li>  
      ))}  
    </ul>  
  );  
}
```

**NO - Istruzioni JavaScript (Statements):** Istruzioni come `if`, `for`, `while`, ecc., non possono essere direttamente inserite all'interno del JSX.

**NO - Nomi di attributi non standard:** JSX non permette l'uso di attributi arbitrari simili a HTML che non sono riconosciuti da React. Devi utilizzare solo quelli che React riconosce, come `className`, `style`, ecc.

**NO - Commenti JavaScript tradizionali:** I commenti JavaScript all'interno del JSX devono essere racchiusi in parentesi graffe e utilizzare la sintassi del commento del blocco `/* */`.

jsx

React

# JSX

Puliamo tutto App.js e proviamo a scrivere la data di oggi.

```
import './App.css';
const saluto=<h3>Ciao bello</h3>
function getDate(date){
  return date.toLocaleDateString() + " " + date.toLocaleTimeString()
}
function App() {
  return (
    <div className="App">
      <h1>Ciao</h1>
      <h2>
        {
          new Date().toLocaleDateString() + " " + new Date().toLocaleTimeString()
        }
      </h2>
      <h2>{getDate(new Date())}</h2>
      <h2>{getDate(new Date())}</h2>
      <h2>{getDate(new Date())}</h2>

      {saluto}
    </div>
  );
}

export default App;
```

# JSX

Se volessimo far correre l'orologio potremmo andare su index.js e mettere un setTimeout ogni secondo in modo che App venga rendereizzata in questo intervallo.

Otterremo il nostro scopo ma ovviamente non è il metodo corretto. Nel coso delle prossime slide vedremo come fare utilizzando la reattività di React attraverso l'uso dei componenti e degli Hook

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
function renderApp(){

  root.render(
    <React.StrictMode>
      <App />
    </React.StrictMode>,
    document.getElementById('root')
  );
}
setInterval(renderApp,1000);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

React

**React.StrictMode** è un wrapper component che aiuta a identificare problemi potenziali nella tua applicazione **solo durante lo sviluppo**. Non fa nulla in produzione.

**React.StrictMode** è un wrapper component che aiuta a identificare problemi potenziali nella tua applicazione solo durante lo sviluppo. Non fa nulla in produzione.

# Componente

Un componente React è una unità indipendente di codice che rappresenta una parte di un'interfaccia utente (UI). Esso è essenzialmente un pezzo di codice riutilizzabile che può avere uno stato proprio, ricevere dati da fonti esterne attraverso le props (proprietà) e può rendere l'interfaccia utente basata su queste informazioni.

# Caratteristiche Componente

**Riutilizzabile:** Una delle principali motivazioni dietro l'uso dei componenti è la loro riutilizzabilità. Puoi definire un componente una volta e poi utilizzarlo in molteplici luoghi all'interno della tua applicazione.

**Encapsulation:** Ogni componente ha la sua logica e la sua struttura, rendendo facile ragionare su di esso come un'entità singola.

**Stato e Proprietà (Props):** I componenti possono ricevere input esterni tramite "props" e possono avere uno "stato" interno. Quando lo stato o le props di un componente cambiano, il componente può ricaricarsi per riflettere queste modifiche.

# Caratteristiche Componente

**Ciclo di Vita:** I componenti React hanno vari "metodi del ciclo di vita" che consentono di eseguire determinate azioni in diverse fasi della vita di un componente, come quando viene montato sul DOM o quando viene aggiornato.

**JSX:** I componenti React sono spesso scritti usando JSX (JavaScript XML), che permette di scrivere la struttura UI in un modo che assomiglia all'HTML, ma è integrato direttamente con il codice JavaScript.

**Componibili:** I componenti possono essere annidati all'interno di altri componenti, permettendo di costruire interfacce utente complesse da componenti più piccoli e gestibili.

# Caratteristiche Componente

**Funzionali vs Classi:** Inizialmente, React aveva componenti basati su classi, ma con l'introduzione degli Hooks in React 16.8, la creazione di componenti funzionali è diventata più popolare e potente. I componenti funzionali sono generalmente più concisi e permettono di utilizzare le funzionalità dello stato e del ciclo di vita in un modo più semplice e diretto.

**Virtual DOM:** Quando lo stato o le props di un componente cambiano, React crea una rappresentazione virtuale del DOM (Virtual DOM). Successivamente, confronta questa versione con la versione precedente e calcola la differenza (chiamata "diffing"). Infine, aggiorna solo le parti effettive del DOM che devono essere modificate, piuttosto che ricaricare l'intero DOM. Questo rende le aggiornamenti dell'interfaccia utente molto efficienti.

# Creare un Componente

In sintesi, React si basa sulla creazione e combinazione di componenti. Questi componenti sono riutilizzabili e reattivi, e approfondiremo in seguito cosa significhi "reattivo".

In definitiva, con React, creiamo elementi HTML personalizzati e li combiniamo per realizzare un'interfaccia utente. Ma basta con la teoria, è ora di immergerci nel codice e iniziare a costruire questi componenti.

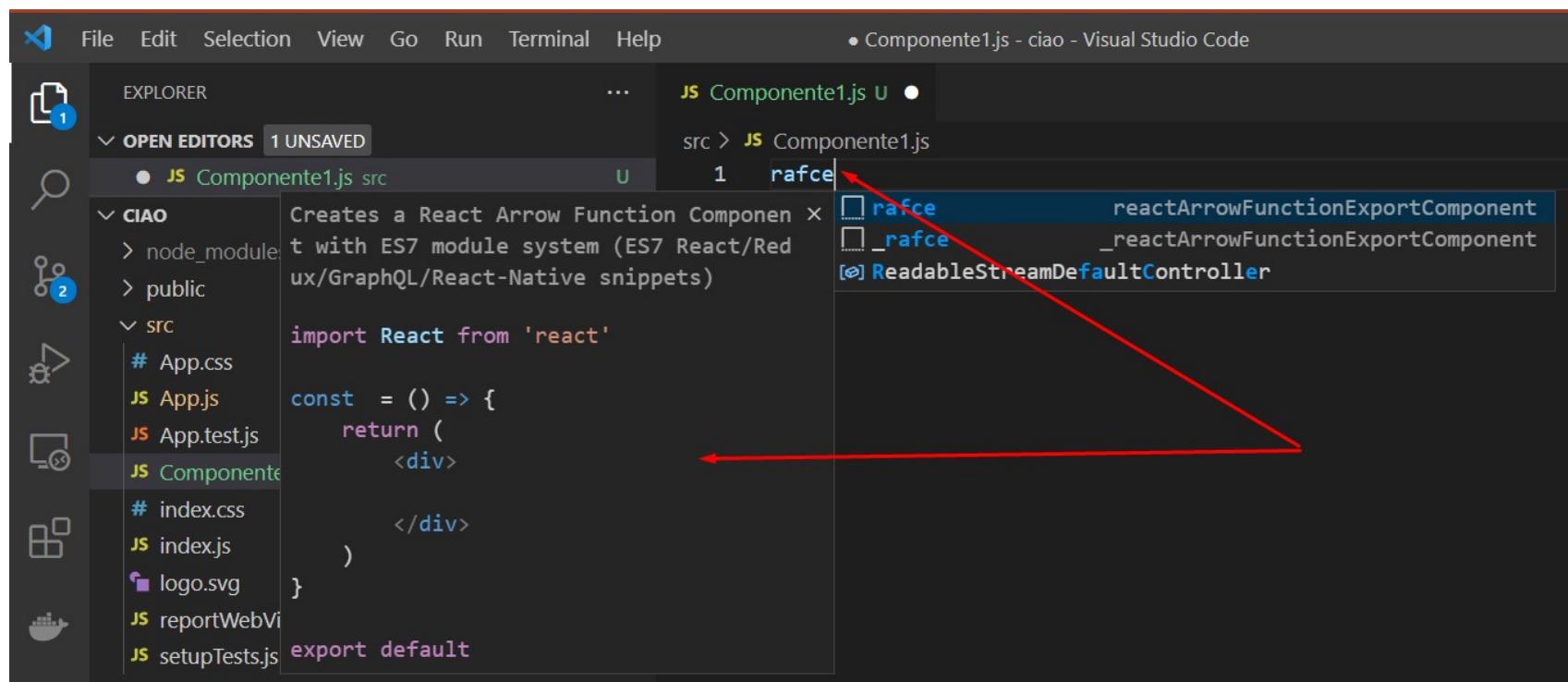
Per creare un nuovo componente bisogna creare un nuovo file .js all'interno della cartella src, ad esempio chiamiamolo componente1.js.

Prima di scrivere qualcosa sfruttiamo il nostro visual code studio ed andiamo ad importare **l'estensione ES7 React** ed installiamo il plugin.

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Active Editor:** App.js - ciao - Visual Studio Code
- Left Sidebar:** Icons for File, Find, Replace, Debug, and Snippets.
- Search Bar:** EXTENSIONS: MARKETPLACE, containing the search term "es7 react".
- Search Results:**
  - ES7 React/Redux/G...** by dsznajder, version 3.1.1, 2.9M installs, 4.5 stars. Description: Simple extensions for React, Redux and Gr... **Install**
  - React Native Tools** by Microsoft, version 1.5.1, 2.1M installs, 4 stars. Description: Debugging and integrated commands for ... **Install**
  - React-Native/React/...** by EQuimper, version 2.0.0, 665K installs, 5 stars. Description: Code snippets for React-Native/React/Red... **Install**
  - Kite AutoComplete...** by Kite, version 0.147.0, 2.5M installs, 3.5 stars. Description: AI code completions for all languages, inte... **Install**
- Code Editor:** Shows the content of App.js with syntax highlighting and line numbers 1 through 13. A red arrow points from the search bar to the "Install" button for the first extension.

Il comando più usato del plugin appena installato è rafce (**R**eact **A**rror **F**unction **E**xport **C**omponent) che ci facilita la scrittura dello scheletro di un componente :



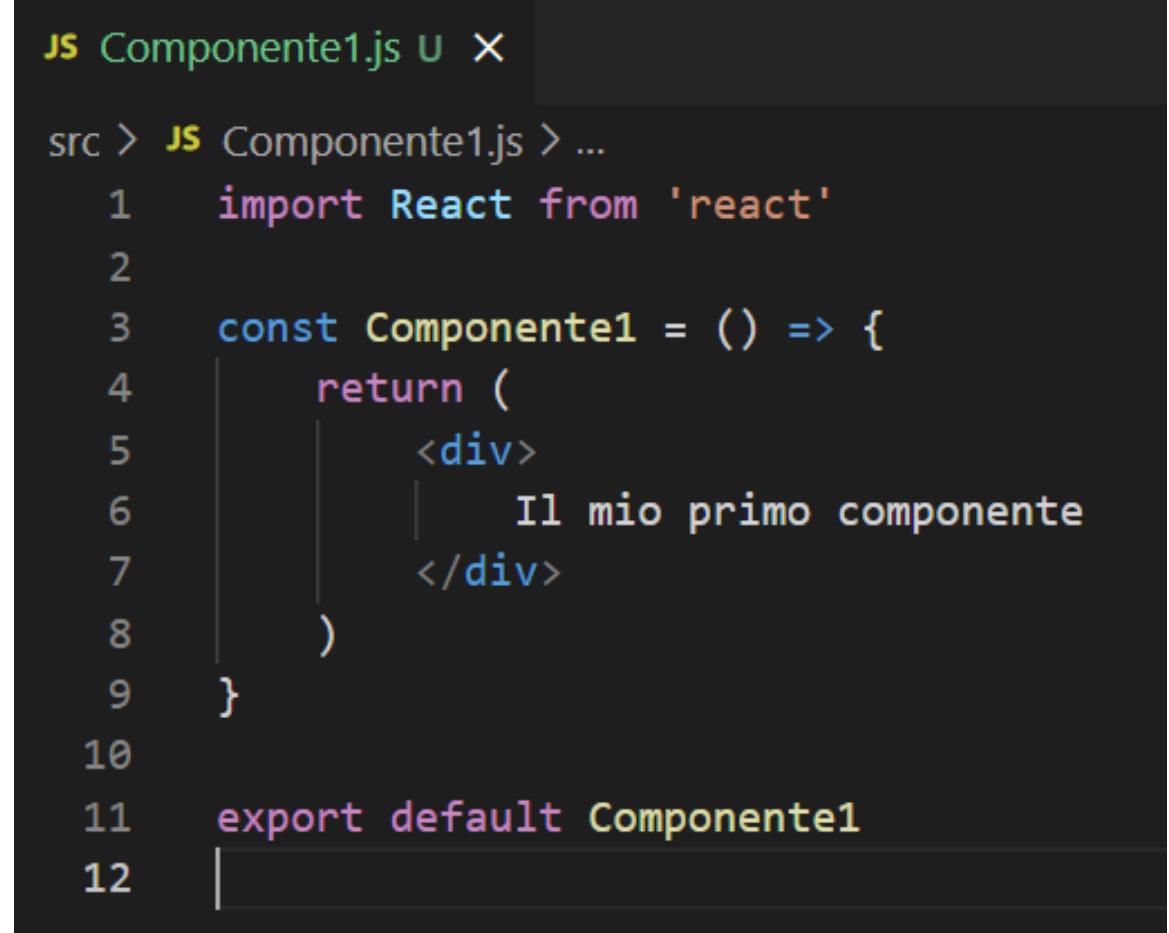
The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Explorer:** Shows a file tree with a folder named "CIAO" containing "node\_modules", "public", and "src". Inside "src", there are files: "# App.css", "JS App.js", "JS App.test.js", "JS Componente1.js", "# index.css", "JS index.js", "logo.svg", "JS reportWebVi", and "JS setupTests.js".
- Editor:** The file "JS Componente1.js" is open. The code starts with an import statement and a function component definition:

```
import React from 'react'

const = () => {
  return (
    <div>
      </div>
  )
}
```
- Completion Panel:** A completion panel is displayed at the top right of the editor, triggered by the "rafce" command. It shows three suggestions:
  - rafce reactArrowFunctionExportComponent
  - \_rafce \_reactArrowFunctionExportComponent
  - [?] ReadableStreamDefaultController

Una volta creato il componente scriviamo una semplice frase all'interno del div :



The screenshot shows a code editor window with a dark theme. The file is named "Componente1.js". The code defines a simple React component:

```
JS Componente1.js U X
src > JS Componente1.js > ...
1 import React from 'react'
2
3 const Componente1 = () => {
4     return (
5         <div>
6             Il mio primo componente
7         </div>
8     )
9 }
10
11 export default Componente1
12 |
```

Una volta creato il nostro primo componente possiamo richiamarlo all'interno del componente principale App.js importando il componente appena creato ed utilizzandolo scrivendo il tag con il nome scelto:

```
JS Componente1.js U JS App.js M X JS index.js # App.css
src > JS App.js > ...
1  import logo from './logo.svg';
2  import './App.css';
3  import Componente1 from './Componente1';
4
5  function App() {
6    return (
7      <div className="App">
8        <h1>Componente Principale</h1>
9        <Componente1/>
10     </div>
11   );
12 }
13
14 export default App;
```

**ATTENZIONE** : Ricordarsi che è possibile scrivere in html <Componente1></Componente1> oppure allo stesso modo è possibile scrivere <Comonente1/> cioè scrivere un unico tag con la chiusura alla fine... È la stessa cosa !

Altra cosa importante da ricordare è che i componenti **DEVONO** sempre tornare qualcosa, quindi ogni componente deve terminare sempre con **return** .....

**Tornando al nostro esempio possiamo creare un componente Clock da riutilizzare ogni volta che vogliamo**

```
const Clock = () => {
  const date = new Date();
  return <h2>Today is {date.toLocaleDateString() + ' ' + date.toLocaleTimeString()}</h2>;
};

export default Clock;
```

# **Componente Principale**

Il mio primo componente

# Sintassi JSX

Riprendiamo il nostro componente appena scritto, il codice all'interno del **return** sembra un semplice codice HTML ma non dimentichiamoci che siamo all'interno di javascript e che il linguaggio che usiamo è JSX

```
import logo from './logo.svg';
import './App.css';
import Componente1 from './Componente1';

function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      <Componente1/>
    </div>
  );
}

export default App;
```

JSX è il linguaggio che definisce il markup HTML da restituire per il rendering visuale all'interno della pagina. Ci sono alcune regole da considerare, quando si utilizza la sintassi JSX: perché JSX funzioni, è necessario **wrappare** tutti gli elementi in un singolo tag. Ad esempio:

```
// NON VALIDO
const invalidJSX = <em>Hello</em>, <strong>World</strong>

// VALIDO
const validJSX = <div>
  <em>Hello</em>, <strong>World</strong>
</div>
```

Ad ogni modo react ci da la possibilità di avere i «tag» non wrappati direttamente in un tag «html». In questo caso dovremmo scrivere come in immagine usando React.Fragment (<>... </>)

```
src > js Componente1.js > [o] default
  1 import React from "react";
  2
  3 const Componente1 = () => {
  4   return (
  5     <React.Fragment>
  6       <h2>Ciao</h2>
  7       <h3>Cio Ciao</h3>
  8     </React.Fragment>
  9   );
 10 };
 11
 12 export default Componente1;
 13
```

Altra regola importante da tenere in considerazione che per richiamare una classe CSS dobbiamo usare la parola chiave **className** invece di class (questo perché in javascript, class è una parola riservata), inoltre è obbligatorio **chiudere** tutti i tag anche se sono singoli e non hanno apertura e chiusura, per esempio  
`<br></br>`

# Componenti Innestati

Vediamo ora come è possibile inserire più componenti in un unico componente.

```
const Componente1 = () => {
  return (
    <div>
      Il mio primo componente
      <Anagrafica/>
      <Messaggio/>
    </div>
  )
}

const Anagrafica = () => {
  return (
    <h2>Il mio nome è Riccardo</h2>
  )
}

const Messaggio = () => {
  return (
    <p>Benvenuti a tutti</p>
  )
}

export default Componente1
```

# Componenti Innestati

E' possibile esportare più componenti da un unico file? Certamente si

```
import React from 'react'
const Componente1 = () => {
  return (
    <div>Componente1</div>
  )
}
export const Anagrafica = () => {
  return (
    <div>Anagrafica</div>
  )
}

export const Messaggio = () => {
  return (
    <div>Messaggio</div>
  )
}

export default Componente1
```

```
import Componente1, {
  Anagrafica, Messaggio } from
  './percorso/del/file';
```

```
function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      <Componente1/>
      <Componente1/>
      <Componente1/>
      <Componente1/>
    </div>
  );
}

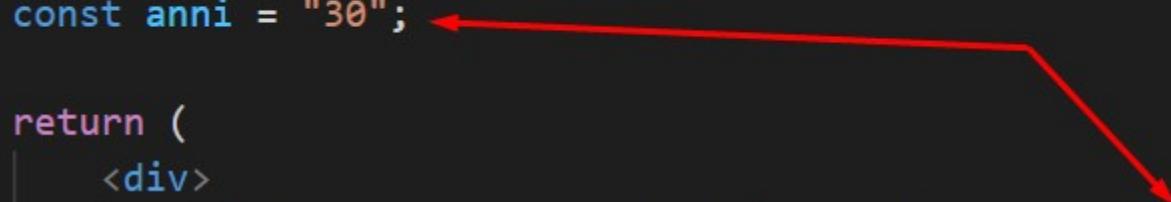
export default App;
```



# Variabili in un componente

All'interno di un componente possiamo dichiarare e utilizzare variabili facendo poi riferimento all'interno di JSX attraverso le parentesi graffe. Ad esempio :

```
const Componente1 = () => {  
  
    const anni = "30"; ←  
  
    return (  
        <div>  
            Il mio primo componente l'ho realizzato a {anni} anni  
            <Anagrafica/>  
            <Messaggio/>  
        </div>  
    )  
}
```



# Style inline

Un'altra cosa a cui prestare attenzione è lo style inline in quanto è possibile dichiarare l'attributo style all'interno di un tag, ma a differenza dell'html, in JSX bisogna mettere direttamente l'oggetto CSS in questo modo :

```
style = {color : red}
```

Visto che la graffa viene usata per richiamare variabili, nel caso del css viene messa una doppia graffa:

```
style = {{ color:red }}
```

```
const Anagrafica = () => {
  return (
    <h2 style={{'color':'#ff0000' }}>Il mio nome è Riccardo</h2>
  )
}
```

# Il Props Object

Vediamo ora come possiamo passare dei parametri ai nostri componenti. I parametri che passiamo ad un componente vengono chiamati per convenzione **props**, si consiglia di usare questa convenzione in modo da rendere il nostro codice il più leggibile possibile.

Per passare un parametro è sufficiente passare props in ingresso alle parentesi del componente in questo modo :

```
const Componente1 = (props) => {
    const anni = "30";

    return (
        <div>
            Il mio primo componente {props.nome} l'ho realizzato a {anni} anni
            <Anagrafica/>
            <Messaggio/>
        </div>
    )
}
```

```
function App() {
    return (
        <div className="App">
            <h1>Componente Principale</h1>
            <Componente1 nome="di prova 1"/>
            <Componente1 nome="di prova 2"/>
            <Componente1 nome="di prova 3"/>
            <Componente1 nome="di prova 4"/>
        </div>
    );
}
```

# Props Children

Il children dei props è il codice che mettiamo tra i tag di apertura e chiusura per esempio

```
<Componente1>Questo è il testo che viene inserito in children</Componente1>
```

```
{props.nome} {props.cognome} di anni {props.anni}  
<p>{props.children}</p>
```

Nella maggior parte dei casi, quando passo il parametro ad un componente, difficilmente mi trovo a passare un solo parametro, di solito sono più di uno. In questo caso non cambia nulla. Ad esempio nel nostro componente oltre a passare il nome, passiamo anche il cognome :

```
function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      <Componente1 nome="riccardo" cognome="cattaneo" eta="30"/>
      <Componente1 nome="mario" cognome="rossi" eta="35"/>
      <Componente1 nome="lucia" cognome="verdi" eta="40"/>
      <Componente1 nome="anna" cognome="casale" eta="38"/>
    </div>
  );
}
```

```
const Componente1 = (props) => {  
  
    return (  
        <div>  
            Il componente è di {props.nome} {props.nome} ed ho {props.anni} anni  
            <Anagrafica/>  
            <Messaggio/>  
        </div>  
    )  
}  
}
```

Ma volendo, quando devo chiamare un componente, posso anche passare i parametri tutti insieme attraverso lo **Spread Operator**. Riconoscibile dai **tre punti ...**

# Spread Operator

```
const primaPersona = {  
    nome : "riccardo",  
    cognome : "cattaneo",  
    eta : "30"  
}  
  
const secondaPersona = {  
    nome : "mario",  
    cognome : "rossi",  
    eta : "35"  
}  
  
function App() {  
    return (  
        <div className="App">  
            <h1>Componente Principale</h1>  
            <Componente1 {...primaPersona}>/>  
            <Componente1 {...secondaPersona}>/>  
            <Componente1 nome="lucia" cognome="verdi" eta="40"/>  
            <Componente1 nome="anna" cognome="casale" eta="38"/>  
        </div>  
    );  
}
```

# Torniamo all'esempio del nostro Clock e passiamo come parametri il country e il timezone

```
1  const Clock = ({ country, timezone }) => {
2    const t = Date.now() + 3600 * timezone * 1000;
3    const date = new Date(t);
4    console.log(date, t, timezone);
5    return (
6      <h2>
7        | In {country} is {date.toLocaleDateString() + ' ' + date.toLocaleTimeString()}
8      </h2>
9    );
10  };
11
12  export default Clock;
13
14
```

# Array in JSX

Negli esempi precedenti abbiamo usato singoli oggetti. Ma come sappiamo normalmente si usano gli array nella programmazione, che non sono altro che un insieme di oggetti racchiusi nella stessa variabile.

Proviamo a modificare il nostro codice e trasformiamo i nostri due oggetti in un array di oggetti :

```
const persone = [ {  
    nome : "riccardo",  
    cognome : "cattaneo",  
    eta : "30"  
}, {  
    nome : "mario",  
    cognome : "rossi",  
    eta : "35"  
}];
```

Se proviamo a fare il render dell'oggetto all'interno del nostro componente notiamo che ci restituisce un errore. Questo perché stiamo cercando di visualizzare un oggetto.

Error: Objects are not valid as a React child (found: object with keys {nome, cognome, eta}). If you meant to render a collection of children, use an array instead.

Questo avviene perché se vogliamo fare il render di un oggetto dobbiamo accedere ad ogni singola proprietà, non possiamo passargli tutto l'oggetto.

A questo punto ci viene in aiuto il metodo map che ci consente di prendere un array, parsando ogni singolo elemento.

# map method

Il metodo **map** può essere richiamato direttamente sulla variabile che per noi rappresenta l'array, nel nostro caso **persone.map()**.

Questo metodo prende in ingresso il nome di una variabile che per noi rappresenta una variabile di appoggio dove viene memorizzato di volta in volta ogni singolo elemento dell'array, e tramite una arrow function posso gestire l'output in questo modo :

```
const persone = [
  nome : "riccardo",
  cognome : "cattaneo",
  eta : "30"
},{
  nome : "mario",
  cognome : "rossi",
  eta : "35"
];

function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      {
        persone.map(pers => {
          return <h1>{pers.cognome}</h1>;
        })
      }
    </div>
  );
}
```

# Il Key Attribute

Se apriamo la nostra console possiamo notare che c'è un errore...

```
✖ Warning: Each child in a list should have a unique "key" prop.  
Check the render method of `App`. See https://reactjs.org/link/warning-keys for more information.  
  at h1  
  at App
```

Questo perché ogni elemento di un array ritornato dinamicamente DEVE avere un proprio attributo **key**.

Si può fare in due modi : il primo è quello di «sfruttare» l'indice che ci viene fornito automaticamente dalla funzione map. Se infatti ci aiutiamo con il nostro editor visual studio code, appena scriviamo il nostro metodo map, ci suggerisce che è possibile gestire anche il nostro indice (index) in questo modo :

```
function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      {
        persone.map((pers,index) => {
          return <h1 key={index}>{pers.cognome}</h1>;
        })
      }
    </div>
  );
}
```



React

Oppure un modo alternativo, e pure più corretto è quello di aggiungere un campo id all'interno del nostro oggetto e usare quello come key del nostro array :

```
const persone = [
  {
    id : 1,
    nome : "riccardo",
    cognome : "cattaneo",
    eta : "30"
  },
  {
    id : 2,
    nome : "mario",
    cognome : "rossi",
    eta : "35"
  }
];

function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      {
        persone.map((pers) => {
          return <h1 key={pers.id}>{pers.cognome}</h1>;
        })
      }
    </div>
  );
}

React
```

Per rendere il nostro codice più pulito, in considerazione del fatto che non abbiamo database dove andare a memorizzare i dati, si consiglia di memorizzare i nostri dati in un file .js ed importarlo di volta in volta all'interno dei nostri componenti.

```
JS databasepersone.js > [Ø] default
const persone = [
    id : 1,
    nome : "riccardo",
    cognome : "cattaneo",
    eta : "30"
},{ 
    id : 2,
    nome : "mario",
    cognome : "rossi",
    eta : "35"
}];

export default persone;
```

```
import logo from './logo.svg';
import './App.css';
import Componente1 from './Componente1';
import persone from './databasepersone';

function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      {
        persone.map((pers) => {
          return <h1 key={pers.id}>{pers.cognome}</h1>;
        })
      }
    </div>
  );
}

export default App;
```

# Eventi

Il primo gestore di eventi che andiamo a vedere è **onClick** che può essere applicato a qualsiasi tag anche se il più usato è il <button>. Ad esempio :

```
function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      {
        persone.map((pers) => {
          return <h1 key={pers.id}>{pers.cognome}</h1>;
        })
      }
      <button onClick={() => alert('invia')}>INVIA</button>
    </div>
  );
}

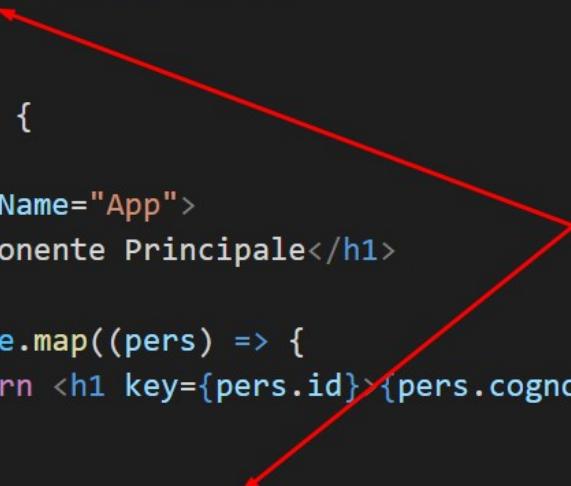
export default App;
```



In questo primo esempio abbiamo scritto la nostra arrow function direttamente nel tag, ma possiamo anche creare un funzione con il codice e richiamarla al verificarsi dell'evento :

```
const invio = () => {
  alert('sto inviando i dati');
}

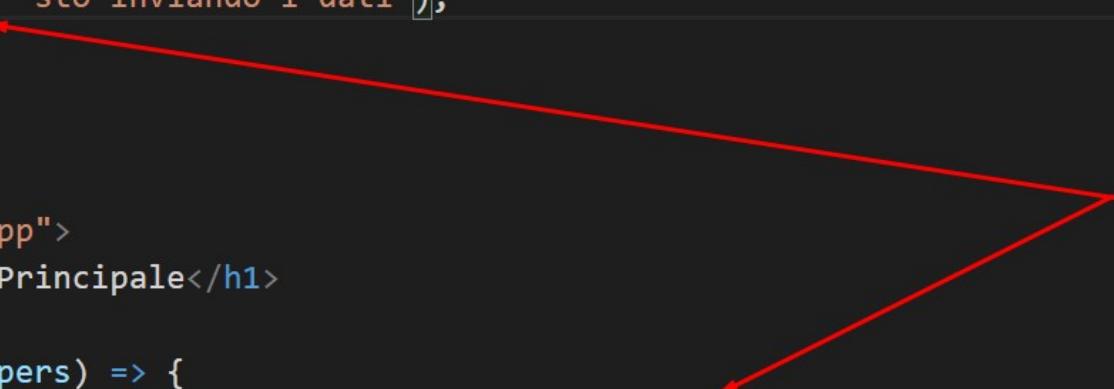
function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      {
        persone.map((pers) => {
          return <h1 key={pers.id}>{pers.cognome}</h1>;
        })
      }
      <button onClick={ invio }>INVIA</button>
    </div>
  );
}
```



Se invece vogliamo passare dei parametri alla funzione sappiamo già come fare :

```
const invio = (nominativo) => {
|   alert(nominativo + ' sto inviando i dati');
}

function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      {
        persone.map((pers) => {
          return <h1 key={pers.id} onClick={() => invio(pers.nome)}>{pers.cognome}</h1>;
        })
      }
      <button>INVIA</button>
    </div>
  );
}
```



Un altro evento simile a onClick è **onMouseOver** per il quale valgono le stesse regole viste per l'onclick.

```
const invio = (nominativo,cognome) => {
  console.log(nominativo + cognome + ' sto inviando i dati');
}

function App() {
  return (
    <div className="App">
      <h1>Componente Principale</h1>
      {
        persone.map((pers) => {
          return <h1 key={pers.id} onMouseOver={() => invio(pers.nome,pers.cognome)}>
            {pers.nome} {pers.cognome}
          </h1>
        })
      }
      <button>INVIA</button>
    </div>
  );
}
```

# Callback function

Per far ein modo che un componente figlio possa passare parametri al padre possiamo usare le call back function come segue

```
function App() {
  const stampaNome=(nome)=>{
    alert(nome)
  }
  return(
    <>
    <Componente1 onStampa={stampaNome}></Componente1>
    <br/>
    </>
  )
}

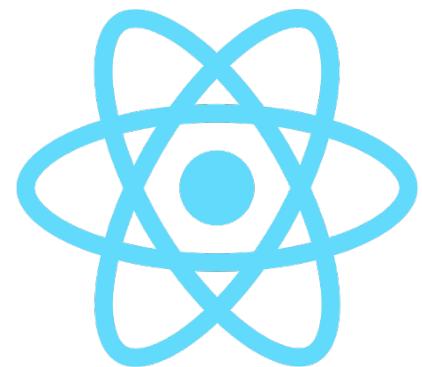
import React from 'react'
const Componente1 = ({onStampa}) => {
  return (
    <div>
      <h2>Componente1</h2>
      <button onClick={()=>onStampa('pippo')}>Clicca</button>
    </div>
  )
}
```

# Esercizi

- 1- Scrivere un componente Persona che mostri a video i dati anagrafici contenuti in un oggetto persona. Formattare il layout con bootstrap
- 2- Scrivere un componente Tabellina che stampi la tabellina di un numero, compreso tra 1 e 10 a vostro piacimento.
- 3- Scrivere un componente Stampanumeri che stampa i numeri da 0 a 10 ;
- 4- Scrivere un programma che conta da 0 a 20 con passo 2 e stampa i numeri ottenuti (0,2,...,20) ;
- 5- Creare un componente Biblioteca che visualizza una lista di libri e permette agli utenti di aggiungere un nuovo libro alla lista.  
**Componente Biblioteca:** Il componente principale che ospita gli altri componenti.  
**Componente BookList:** Un componente che mostra la lista dei libri.  
**Componente AddBookForm:** Un componente che al momento mostra solo la scritta ‘form inserimento libro’ con un pulsante che al click mostra un alert(‘libro inserito’)  
**Simulazione di Aggiornamento:** Utilizzare una funzione per aggiungere un libro alla lista.  
Nota Bene: al momento non ci sarà il render lato front end ma andremo a stampare tutto in console

# React

React



# Hooks

Prima di vedere cosa sono gli Hooks e come funzionano, andiamo a chiarire cosa fa react al cambio valore di una variabile. Per fare un esempio andiamo a creare un componente e al suo interno dichiariamo una variabile con valore ‘ciao’ ed una funzione che cambia il valore a tale variabile in ‘arrivederci’ ed all’interno del rendering creare un bottone che al verificarsi l’evento onClick richiama la funzione cambiaValore :

```
function App() {  
  let saluto = "ciao";  
  
  const cambioSaluto = () => {  
    saluto = "arrivederci";  
    console.log(saluto);  
  }  
  return (  
    <div className="App">  
      <h1>Componente Principale</h1>  
      <h2>{saluto}</h2>  
      <button onClick={cambioSaluto}>Cambia</button>  
    </div>  
  );  
}
```

Se vediamo il risultato all'interno della console possiamo constatare che effettivamente il valore della variabile è cambiato in memoria, ma non ha aggiornato il rendering del codice visualizzato a video.

Abbiamo bisogno quindi di «aggiornare» il nostro componente, per farlo ci vengono in aiuto gli Hooks e più precisamente **useState**.

# useState

Quando creiamo un componente, non abbiamo a disposizione costruttori o altri metodi per inizializzare lo **stato** quindi dobbiamo utilizzare un'altra tecnica: gli **hook**.

Nel caso specifico, il metodo **useState** è l'hook da utilizzare. Questo **metodo** accetta in input un valore che rappresenta il valore iniziale da dare a una variabile dello stato e restituisce in output un oggetto con **una variabile che rappresenta lo stato** e un **metodo da invocare per modificare la variabile nello stato**.

<https://it.reactjs.org/docs/hooks-intro.html#gatsby-focus-wrapper>

**useState** ha due funzionalità principali : ci permette di avere un render della variabile, in modo che lo stato viene mutato al mutare della variabile, ma soprattutto di tenere traccia dello stato che viene mutato.

Per prima cosa dobbiamo importare la funzione useState from react in questo modo :

```
import { useState } from 'react';
import './App.css';
```

```
function App() {
```

```
    let saluto = "ciao";
```

La prima cosa che andiamo a fare è stampare tramite console.log il valore di useState e possiamo vedere che si tratta di una funzione :

```
import { useState } from 'react';
import './App.css';

function App() {
    console.log(useState);
```



```
f useState(initialState) {
    var dispatcher = resolveDispatcher();
    return dispatcher.useState(initialState);
}
```

Navigated to <http://localhost:3000/>

Se proviamo a passare alla funzione useState un valore di qualsiasi tipo (un oggetto, una stringa, un numero o qualsiasi tipo) vediamo che **ci ritorna un array** composto dal valore passato in ingresso ed in più una funzione :

```
import { useState } from 'react';
import './App.css';

function App() {

  const value1 = useState()[0];
  const value2 = useState()[1];

  console.log(value1,value2);
```

React

```
undefined f dispatchAction(fiber, queue, action) {
  if (typeof arguments[3] === 'function') {
    error("State updates from the useState() and useReducer() Hooks don't support the " + 'secon...')
```

Dopo questo primo esempio andiamo a creare un **useState** reale, riproponendo lo stesso esempio di prima, e cioè creare un bottone che chiama una funzione per cambiare valore alla variabile :

```
function App() {  
  
  const salutami = useState('buongiorno');  
  
  console.log('primo ' + salutami[0]);  
  console.log('secondo ' + salutami[1]);  
  
  const cambioSaluto = () => {  
    salutami[1]("arrivederci");  
    console.log(salutami[0]);  
  }  
  return (  
    <div className="App">  
      <h1>Componente Principale</h1>  
      <h2>{salutami[0]}</h2>  
      <button onClick={cambioSaluto}>Cambia</button>  
    </div>  
  );  
}
```

# Alcune Regole

- La prima regola da ricordare è che tutti gli Hook **DEVONO** iniziare con **use**, anche quelli che creeremo noi.
- Il componente in cui usiamo gli Hook **DEVE** avere il nome con la prima lettera **maiuscola**.
- Gli Hook **DEVONO** essere chiamati **all'interno** del nostro componente

# useState con Array

Prima di vedere come si comporta useState con un array andiamo a creare una cartella dove creiamo un file chiamato data.js ed andiamo a mettere un array di oggetti con due campi : id e nome e popoliamola a piacimento con qualche nominativo (simulando la chiamata ad un database). Ricordiamo anche di importare useState dalla libreria di react :

```
src > data > JS data.js > [?] anagrafica
```

```
1  export const anagrafica = [
2      {id : 1, nome : "Mario"},
3      {id : 2, nome : "Luca"},
4      {id : 3, nome : "Anna"},
5      {id : 4, nome : "Giuseppe"},
6      {id : 5, nome : "Francesco"}
7 ]
```

```
import { useState } from 'react';
import './App.css';
```

Se ora proviamo a passare alla funzione useState l'array anagrafiche appena importata possiamo verificare che il risultato è sempre lo stesso, la funzione ci torna l'oggetto passato in ingresso ed una funzione :

```
import { useState } from 'react';
import { anagrafica } from './data/data';
import './App.css';

function App() {

    const persone = useState(anagrafica);

    console.log('primo ' + persone[0]);
    console.log('secondo ' + persone[1]);
```

# Destruzione

Un modo alternativo di gestire il ritorno di una funzione, se il ritorno è composto da più oggetti è possibile «destrutturarli» in questo modo, ma è solo per comodità :

```
const persone = useState(anagrafica);
// destruzione
const [anag, setAnag] = useState(anagrafica);

console.log('primo ' + anag);
console.log('secondo ' + setAnag);
//console.log('primo ' + persone[0]);
//console.log('secondo ' + persone[1]);
```

Arrivati a questo punto abbiamo un array che se vogliamo stamparlo a video già sappiamo come fare.... Usando la funzione map in questo modo : l'unica differenza rispetto a prima che aggiungiamo un bottone vicino ad ogni oggetto ...

```
return (
  <div className="App">
    <h1>Componente Principale</h1>

    {anag.map( el => {
      const {id, nome} = el;
      return(
        <div key={id}>
          <span>{nome}</span>
          <button>Elimina</button>
        </div>
      )
    })}
  
```

Il nostro obiettivo è quello di eliminare una voce dall'array nel momento in cui clicchiamo sul corrispondente bottone Elimina.

Per farlo andiamo a creare una funzione che richiameremo al click, e questa funzione attraverso la funzione filter va a selezionare gli oggetti in base ad una «condizione» passata in ingresso come parametro :

```
const eliminaoggetto = (id) => {
  let nuoviOggetti = anag.filter( el => el.id !== id);
  setAnag(nuoviOggetti);
}

return (
  <div className="App">
    <h1>Componente Principale</h1>

    {anag.map( el => {
      const {id, nome} = el;
      return(
        <div key={id}>
          <span>{nome}</span>
          <button onClick={() => eliminaoggetto(id)}>Elimina</button>
        </div>
      )
    })}
  </div>
);
```

# useState con Oggetto

Se usiamo un oggetto con useState non ci cambia nulla, dobbiamo solo prestare attenzione ad un particolare. Supponiamo di avere un oggetto persona con nome, cognome ed età, e tramite un evento vado a cambiare il valore di un solo elemento dell'oggetto, cosa succede ? Proviamo :

```
const [persona, setPersona] = useState({
  nome : 'Riccardo',
  cognome : 'Cattaneo',
  eta : 30,
});

const compleanno = () => [
  let anni = persona.eta + 1;
  setPersona({
    eta : anni
  });
]

return (
  <div className="App">
    <h1>Componente Principale</h1>
    <h3>{persona.nome}</h3>
    <h3>{persona.cognome}</h3>
    <h3>{persona.eta}</h3>
    <button onClick={compleanno}>Compleanno</button>
  </div>
);
}
```

Se proviamo questo codice ci accorgiamo di due cose... la prima è che «funziona» 😊 la seconda è che, se aggiorniamo un solo dato del nostro oggetto, il resto lo perdiamo, quindi abbiamo il problema di mantenere in qualche modo l'oggetto originale, e qui ci viene in aiuto lo **Spread Operator**, ricordate a cosa serve ? Serve a passare in ingresso ad una funzione o ad un componente un intero oggetto

```
const [persona, setPersona] = useState({  
    nome : 'Riccardo',  
    cognome : 'Cattaneo',  
    eta : 30,  
});  
  
const compleanno = () => {  
    let anni = persona.eta + 1;  
    setPersona([  
        ...persona,  
        eta : anni  
    ]);  
}  
  
return (  
    <div className="App">  
        <h1>Componente Principale</h1>  
        <h3>{persona.nome}</h3>  
        <h3>{persona.cognome}</h3>  
        <h3>{persona.eta}</h3>  
        <button onClick={compleanno}>Compleanno</button>  
    </div>  
);
```

# Torniamo alla data

Ora che abbiamo lo useState vediamo come far muovere l'orologio del nostro esempio iniziale. Creiamo un componente clock.

```
const Clock = () => {
  const date = new Date();
  return <h2>Today is {date.toLocaleDateString() + ' ' + date.toLocaleTimeString()}</h2>;
};

export default Clock;
```

Passiamogli due parametri il timezone e il country

```
1
2  const Clock = ({ country, timezone }) => {
3    const t = Date.now() + 3600 * timezone * 1000;
4    const date = new Date(t);
5    console.log(date, t, timezone);
6    return [
7      <h2>
8        | In {country} is {date.toLocaleDateString() + ' ' + date.toLocaleTimeString()}
9      </h2>
10    ];
11  };
12
13 export default Clock;
14
```

# Torniamo alla data



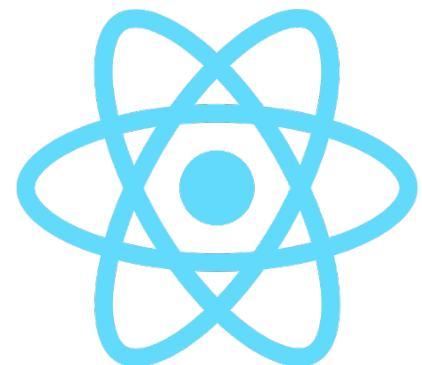
```
js Clock.js M X
...
1 | import { useState } from 'react';
2 |
3 | const Clock = ({ country, timezone }) => [
4 |   const t = Date.now() + 3600 * timezone * 1000;
5 |   const dateIni = new Date(t);
6 |
7 |   const [date, setDate] = useState(dateIni);
8 |
9 |   setTimeout(() => {
10 |     const t = date.getTime() + 3600 * 1000;
11 |     setDate(new Date(t))
12 |   }, 5000);
13 |   return (
14 |     <h2>
15 |       In {country} is {date.toLocaleDateString() + ' ' + date.toLocaleTimeString()}
16 |     </h2>
17 |   );
18 | };
19 |
20 | export default Clock;
21 |
```

# Esercizio

Creare un componente che rappresenta per la mia applicazione un contatore utilizzando useState con valore di default uguale a zero. Sotto al contatore andiamo a creare due bottoni, uno «diminuisci» ed uno «aumenta» che al loro click andranno ad aumentare e a diminuire il valore del contatore.

Utilizziamo bootstrap per rendere gradevole il nostro componente (a nostro piacimento)

# React



# Contatore

Prima di andare avanti andiamo a creare un nuovo componente «contatore» per poi vedere alcune nuove funzionalità di React. Per prima cosa creiamo il componente Contatore e dichiariamo una variabile usando useState e passiamo come valore 0, ed all'interno del nostro componente stampiamo il valore della variabile e aggiungiamo 2 bottoni : aumenta e diminisci.

# Bootstrap

Prima di continuare importiamo le librerie di bootstrap all'interno del nostro progetto. Il modo più semplice è scaricare le librerie da riga di comando :

**npm install - -save bootstrap**

Dopo di ché all'interno del file index.js fare l'import con :

**import 'bootstrap/dist/css/bootstrap.css';**

```
const Contatore = () => {  
  const [contatore, setContatore] = useState(0);  
  
  return (  
    <div className="container">  
      <div className="row justify-content-center align-items-center text-center">  
        <div className="card" style={{'width': '18rem'}}>  
          <h1>{contatore}</h1>  
          <a href="#" class="btn btn-primary">Aumenta</a>  
          <a href="#" class="btn btn-secondary">Diminuisci</a>  
        </div>  
      </div>  
    </div>  
  )  
}
```

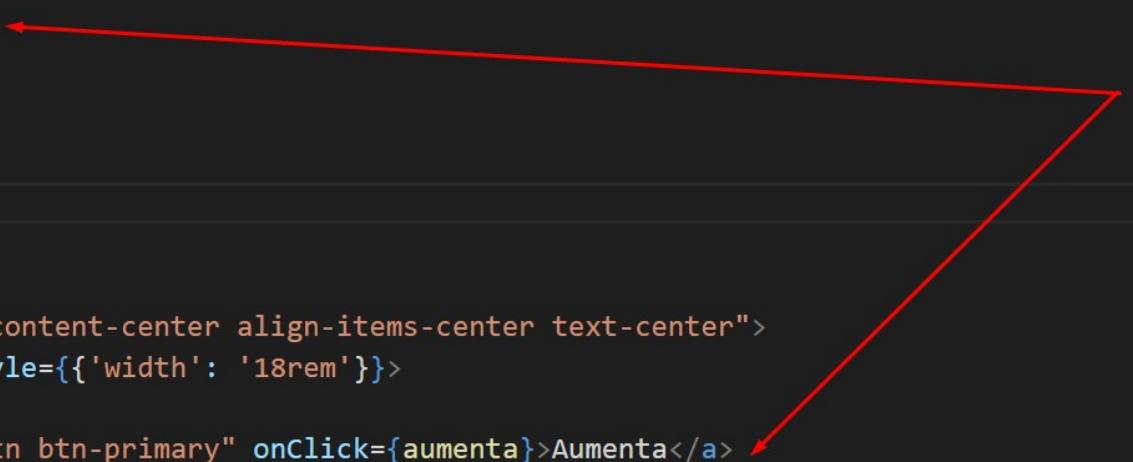
Possiamo ora sviluppare le nostre funzioni in due modi differenti ma che fanno la stessa cosa : il primo modo è mettendo la funzione setContatore direttamente nell'evento onClick tramite una arrow function.

Il secondo modo, quello classico che conosciamo, dove facciamo più passaggi ma il codice risulta più leggibile, cioè tramite una funzione separata. Ricordiamo che la funzione setContatore prende un parametro in **ingresso che rappresenta il valore attuale** del contatore :

```
const Contatore = () => {
  const [contatore, setContatore] = useState(0);

  const aumenta = () => {
    setContatore(valoreAttuale => {
      console.log(valoreAttuale);
      return valoreAttuale + 1;
    })
  }

  return (
    <div className="container">
      <div className="row justify-content-center align-items-center text-center">
        <div className="card" style={{'width': '18rem'}}>
          <h1>{contatore}</h1>
          <a href="#" class="btn btn-primary" onClick={aumenta}>Aumenta</a>
          <a href="#" class="btn btn-secondary" onClick={()=>setContatore(contatore - 1)}>Diminisci</a>
        </div>
      </div>
    </div>
  )
}
```



# useState : return value vs functional

Quando chiamo la funzione useState abbiamo detto che mi torna il valore ed una funzione set. Questa funzione set abbiamo detto che serve per poter gestire ed aggiornare in modo autonomo il valore passato a useState. Ad esempio :

```
const [contatore1, setContatore1] = useState(0);
let contatore2 = 0;
```

Ripetiamo che è la stessa cosa, nel primo caso sto usando la funzione useState nel secondo caso sto valorizzando direttamente la variabile.

Quindi qual è la differenza tra i due ? Che useState è una funzione che torna il suo valore ed in più torna una **funzione che gestisce lo stato** della variabile. Quindi se ad esempio cambio il valore della variabile tramite la sua funzione di ritorno, viene aggiornata automaticamente nel componente (e ricordiamo pure che useState può essere usata solo all'interno di un componente)

```
const [contatore1, setContatore1] = useState(0);
let contatore2 = 0;

const aumenta = () => {
    setContatore1(valoreAttuale => {
        console.log(valoreAttuale);
        return valoreAttuale + 1;
    })
}

const aumenta2 = () => {
    contatore2++;
    console.log(contatore2);
}

const diminuisci2 = () => {
    contatore2--;
    console.log(contatore2);
}

return (
    <div className="container">
        <div className="row justify-content-center align-items-center text-center">
            <div className="card" style={{'width': '18rem'}}>
                <h2>Contatore 1 : {contatore1}</h2>
                <h2>Contatore 2 : {contatore2}</h2>
                <a href="#" class="btn btn-primary" onClick={aumenta}>Aumenta Cont1</a>
                <a href="#" class="btn btn-secondary" onClick={()=>setContatore1(contatore1 - 1)}>Diminuisci Cont1</a>
                <a href="#" class="btn btn-primary" onClick={aumenta2}>Aumenta Cont2</a>
                <a href="#" class="btn btn-secondary" onClick={diminuisci2}>Diminuisci Cont2</a>
            </div>
        </div>
    </div>
)
```

Una volta vista la differenza tra valorizzare una variabile tramite useState e valorizzarla normalmente con il suo valore, vediamo quando una useState cambia valore tramite la sua funzione set che può farlo in 2 modi : tramite il valore o tramite il ritorno di una funzione.

Per fare un esempio andiamo a vedere la differenza sul bottone Aumenta e Diminuisci. Su aumenta cambiamo direttamente il valore, su diminuisci lo cambiamo tramite il ritorno di funzione, detto anche **functional return**.

```
const [contatore1, setContatore1] = useState(0);

const aumenta = () => {
    setContatore1(contatore1+1);
    console.log(contatore1);
}

const diminisci = () => {
    setContatore1(valoreAttuale => {
        console.log(valoreAttuale);
        return valoreAttuale - 1;
    })
}
```

Se proviamo ora il nostro codice effettivamente funziona correttamente in tutte e due i modi. Qual è quindi la differenza tra usare la funzione set **passando direttamente il nuovo valore** oppure usare una **arrow function** che automaticamente prende in ingresso il valore attuale della variabile ?

La differenza è che il **functional return tiene traccia del valore** e lo aggiorna in maniera corretta. Per capirlo modifichiamo il nostro esempio ed aggiungiamo la funzione setTimeout cioè il cambio valore avviene dopo 2 secondi e non subito :

```
const Contatore = () => {  
  const [contatore1, setContatore1] = useState(0);  
  
  const aumenta = () => {  
    setTimeout(function(){  
      setContatore1(contatore1+1);  
      console.log(contatore1);  
    },2000)  
  }  
}
```

In questo caso ad ogni click cambia il valore, ma ogni 2 secondi. Quindi se clicco dopo 2 secondi non ci sono problemi. Ma cosa succede se clicco velocemente per 10 volte ? Succede che non prende tutti e 10 i click in quanto non rimane traccia di tutto quello che è successo mentre attendeva i 2 secondi.

Questo problema viene risolto e viene gestito automaticamente dal **functional return** di useState in questo modo :

```
const aumenta = () => {
  setTimeout(function(){
    setContatore1(valoreAttuale => [
      return valoreAttuale + 1;
    ]);
    console.log(contatore1);
  },2000)
}
```

Per questo motivo normalmente tutti gli useState vengono gestiti con una functional return.

Possiamo usare le functional return anche per implementare dei controlli. Per esempio potremmo volere che il numero non aumenti oltre il 5. In questo caso potremmo fare come segue

```
const aumenta = () => {
  setContatore((oldValue) => {
    if (oldValue + 1 === 5) {
      return oldValue;
    }
    return oldValue + 1;
  });
};
```

# Esercizio Appuntamenti

Creare un'app con una lista di appuntamenti.  
Gli appuntamenti vengono caricati da una lista di oggetti.

```
{  
  id: 4,  
  nome: "Rebecca",  
  stato: "Lorem ipsum"  
  img: «url jps»,  
}
```

L'app farà visualizzare e cancellare gli appuntamenti.

# useEffect

**useEffect** è un Hook che è responsabile di gestire tutte le azioni che avvengono al di fuori del componente.

Per fare un esempio andiamo a creare un nuovo componente ed importiamo lo useEffect così come abbiamo fatto per useState ed anche in questo caso è utilizzabile solo dentro un componente :

```
import React, { useEffect } from 'react'

const EsempioUseEffect = () => {

  useEffect(() => {
    console.log('ho chiamato una useEffect');
  })

  return (
    <div>
      <h1>useEffect</h1>
    </div>
  )
}

export default EsempioUseEffect
```

La **prima regola** che possiamo evincere da questo esempio è che useEffect viene chiamato DOPO che viene fatto il render del componente, praticamente viene chiamato ogni volta che usiamo il nostro componente.

Questo possiamo verificarlo subito andando a stampare dopo dello useEffect un altro log e vediamo che invece lo stampa prima :

```
import React, { useEffect } from 'react'

const EsempioUseEffect = () => {

  useEffect(() => {
    console.log('ho chiamato una useEffect');
  })

  console.log('sono al di fuori di useEffect');

  return (
    <div>
      <h1>useEffect</h1>
    </div>
  )
}

export default EsempioUseEffect
```

▶ XHR finished loading: GET "<http://localhost:3000/79f6b5e...hot-update.json>".

sono al di fuori di useEffect

ho chiamato una useEffect

>

Per fare un esempio andiamo a cambiare il titolo della pagina utilizzando l'oggetto document di javascript (**document.title**).

Ricordiamoci che useEffect viene chiamato ogni volta che il nostro componente subisce un render, quindi se modifichiamo qualcosa al nostro componente verrà automaticamente eseguito anche la funzione useEffect (questo grazie alle useState) :

Riprendiamo l'esempio precedente dove con un click e l'utilizzo di useState vado ad incrementare un numero :

```
const EsempioUseEffect = () => {

  const [valore, setValore] = useState(0); ←

  const aumenta = () => {
    setValore(vecchio => vecchio + 1); ←

  };

  useEffect(() => {
    console.log('ho chiamato una useEffect');
  });

  console.log('sono al di fuori di useEffect');

  return (
    <div>
      <h1>useEffect {valore}</h1>
      <button onClick={aumenta}>Aumenta</button>
    </div>
  )
}

export default EsempioUseEffect
```

# useEffect 15

Aumenta

The screenshot shows the Chrome DevTools Console tab. The left sidebar lists message types: 38 messages, 38 user messages, 3 errors, 3 warnings, 35 info, and No verbose. The main area displays a list of console logs. A red arrow points from the text "useEffect 15" at the top left towards the log entries. The logs consist of repeated messages: "sono al di fuori di useEffect" followed by "ho chiamato una useEffect". There are 15 such pairs of messages. To the right of the logs are several configuration checkboxes: Hide network (unchecked), Preserve log (checked), Selected context only (unchecked), Group similar messages in console (checked), Log XMLHttpRequests (checked), Eager evaluation (checked), Autocomplete from history (checked), and Evaluate triggers user activation (checked). The status bar at the bottom right shows "EsempioUseEffect.js:15" repeated 15 times.

Elements Console Sources Network Performance Memory Security Application Lighthouse

top ▾ Filter Default levels ▾ No Issues

38 messages  
38 user messages  
3 errors  
3 warnings  
35 info  
No verbose

Hide network  
Preserve log  
Selected context only  
Group similar messages in console

sono al di fuori di useEffect  
ho chiamato una useEffect

Log XMLHttpRequests  
Eager evaluation  
Autocomplete from history  
Evaluate triggers user activation

EsempioUseEffect.js:15  
EsempioUseEffect.js:12  
EsempioUseEffect.js:15  
EsempioUseEffect.js:12  
EsempioUseEffect.js:15  
EsempioUseEffect.js:12  
EsempioUseEffect.js:15  
EsempioUseEffect.js:12  
EsempioUseEffect.js:15  
EsempioUseEffect.js:12  
EsempioUseEffect.js:15  
EsempioUseEffect.js:12

All'interno degli Hooks possiamo inserire anche le condizioni ricordando che verranno eseguite solo dopo il render :

```
useEffect(() => {
  console.log('ho chiamato una useEffect');
  if(valore < 1){
    document.title = "Nessun valore";
  }else{
    document.title = "C'è qualcosa..."
  }
});
```

# useEffect secondo parametro

Di default il nostro useEffect verrà eseguito ad ogni render del nostro componente. Ma c'è un modo per decidere quando eseguire il nostro useEffect, e cioè attraverso il secondo parametro che opzionalmente accetta useEffect che è un array.

Se lascio **l'array vuoto** sto dicendo che il nostro useEffect deve essere eseguito una sola volta :

```
useEffect(() => {
  console.log('ho chiamato una useEffect');
  if(valore < 1){
    document.title = "Nessun valore";
  }else{
    document.title = "C'è qualcosa..."
  }
},[]);
```

Oppure posso passare un parametro del componente, serve ad indicare che alla variazione di quel parametro useEffect verrà eseguito :

```
useEffect(() => {
  console.log('ho chiamato una useEffect');
  if(valore < 1){
    document.title = "Nessun valore";
  }else{
    document.title = "C'è qualcosa..."
  }
},[valore]);
```

Niente impedisce, nel rispetto del buon senso, di avere e chiamare molteplici "use effect". Questo scenario non è affatto improbabile, anzi è molto comune. Potrei avere, ad esempio, un "useEffect" che si attiva ogni volta che si aggiorna il valore "street value", e un altro che necessita di essere chiamato sempre. Un ulteriore "use effect" potrebbe essere invocato da me ogni volta che lo ritengo necessario, al quale potrei passare una funzione.

Potrei avere una funzione "Arrow" che fa un semplice log, esclusivamente quando un certo valore non è impostato. È possibile avere un "use effect" che si attiva solo quando c'è una variazione di valore e un altro che si attiva indipendentemente. Entrambi potrebbero essere attivati ogni volta che si verifica un render, a seguito di una variazione di valore.

Se avessi questa configurazione, vedremmo nella console solo il secondo "use effect" che si attiva dopo il primo render, il quale attiva anche il primo "use effect". Dopo ciò, solo il secondo "use effect" verrà invocato. Questo concetto è fondamentale.

```
const [value, setValue] = React.useState(0);

const aumenta = () => {
  setValue((oldValue) => oldValue + 1);
};

const funzio = () => {
  console.log("Secondo use Effect");
};

useEffect(() => {
  console.log("Eccomi, sono use Effect");
  if (value < 1) {
    document.title = `Nessun Messaggio`;
  } else {
    document.title = `Nuovo Messaggi: ${value}`;
  }
}, []);

useEffect(funzio);
```

Un altro aspetto cruciale di "use effect" è la sua capacità di restituire una "**cleanup function**", che ci permette di eseguire operazioni di pulizia. Questa funzione viene definita ogni volta che si esegue "use effect" e viene eseguita prima del successivo render o della successiva invocazione di "use effect".

```
useEffect(() => {
  console.log('ho chiamato una useEffect');
  if(valore < 1){
    document.title = "Nessun valore";
  }else{
    document.title = "C'è qualcosa...";
  }
  return( () => { console.log("Eseguo un po di pulizia") })
},[valore]);
```

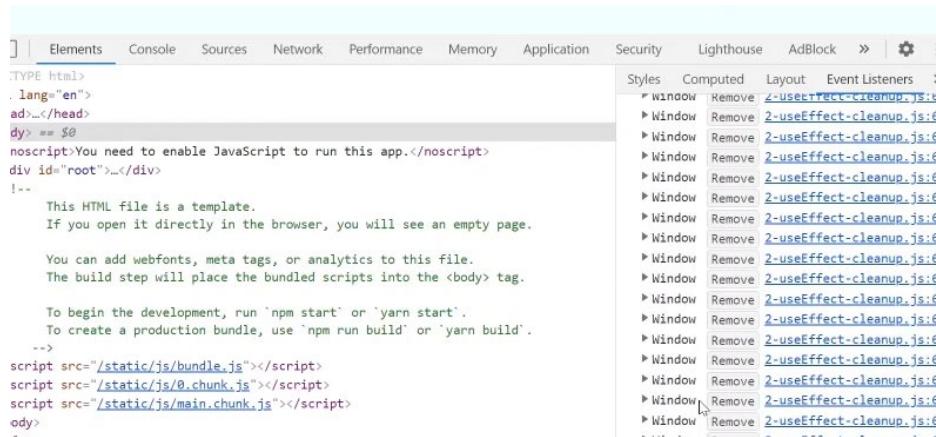
Se immaginiamo di impostare un event listener all'interno del nostro "use effect", che si attiva ogni volta che c'è un aggiornamento di uno specifico stato o elemento, è fondamentale rimuovere quel listener prima della prossima invocazione di "use effect". Altrimenti, se questo stato cambiasse frequentemente, potremmo ritrovarci con l'evento settato numerose volte, compromettendo le performance dell'applicazione. Con la "clean up function", possiamo garantire che l'event listener precedente venga rimosso prima di impostarne uno nuovo.

Vediamo l'esempio più in dettaglio di come usare il return dello

```
const CleanUp = () => {  
  
  const [size, setSize] = useState(window.innerWidth);  
  
  const dimensioneFinestra = () => {  
    setSize(window.innerWidth);  
  };  
  
  /*  
   * Ad ogni resize della pagina Aggiorna la state size e rimuove l'event Listener  
   */  
  useEffect(() => {  
    window.addEventListener("resize", dimensioneFinestra);  
  
  });  
  return (  
    <div  
      className='container w-75 col-6 offset-3 bg-white shadow p-4 mx-auto'  
      style={{ textAlign: "center" }}  
    >  
      <h1> {size} </h1>  
    </div>  
  );  
};  
React
```

Nel componente, abbiamo una semplice visualizzazione di questa dimensione utilizzando Bootstrap. Ogni volta che la dimensione della finestra cambia, la nostra "clean up function" all'interno dell'hook useEffect ci permette di aggiornare lo stato e di rimuovere l'event listener precedentemente aggiunto, garantendo un comportamento ottimale e previene l'aggiunta di listener multipli.

Se osserviamo nella console, vedremo che ogni volta che ridimensioniamo la finestra, viene registrato l'evento. Tuttavia, grazie alla nostra "clean up function", viene sempre mantenuto un solo listener attivo, garantendo prestazioni ottimali e comportamento atteso.



The screenshot shows the Chrome DevTools interface with the 'Elements' tab selected. On the left, the DOM tree displays a template file with various HTML elements like head, body, noscript, and script tags. On the right, the 'Event Listeners' panel is open, showing a list of 18 event listeners registered on the 'Window' object. All these listeners are identical, pointing to the same 'useEffect-cleanup.js:6' function, which corresponds to the clean-up logic in the React component's useEffect hook.

```
TYPE html>
.lang="en">
ad></head>
dy> == $0
noscript>You need to enable JavaScript to run this app.</noscript>
div id="root">...</div>
!!--
This HTML file is a template.
If you open it directly in the browser, you will see an empty page.

You can add webfonts, meta tags, or analytics to this file.
The build step will place the bundled scripts into the <body> tag.

To begin the development, run 'npm start' or 'yarn start'.
To create a production bundle, use 'npm run build' or 'yarn build'.
-->
script src="/static/js/bundle.js"></script>
script src="/static/js/0.chunk.js"></script>
script src="/static/js/main.chunk.js"></script>
ody>
```

Styles	Computed	Layout	Event Listeners
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	
▶ Window	Remove	2-useEffect-cleanup.js:6	

```
useEffect(() => {
  window.addEventListener("resize", dimensioneFinestra);
  return () => {
    window.removeEventListener("resize", dimensioneFinestra);
  };
});
```

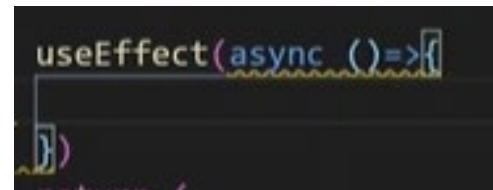
In situazioni più complesse, con molti componenti che possono attaccare event listener in diverse parti dell'applicazione, avere una "clean up function" è buona prassi. Fornisce un controllo completo su ciò che accade, su ciò che viene rimosso e su ciò che rimane attivo. Ci sono molte altre situazioni in cui questo metodo è fondamentale.

# useEffect - Data Fetching

Vediamo come usare useEffect per fetchare dati con fetch o axios.

Creiamo uno useState photos. Andreamo a prendere i dati da jsonplaceholder.

Per fare questo creiamo la funzione async getData e passeremo al nostro useEffect. Dobbiamo creare una funzione a parte perché non **possiamo rendere lo useEffect async**. Il codice dell'immagine quindi non può essere usato



# useEffect - Data Fetching

Di seguito la chiama della funzione con lo useEffect

```
const FetchComponent = () => {  
  
  const [photos, setPhotos] = useState([]);  
  
  const getData = async () => {  
    //const response = await axios.get(url);  
    //setPhotos(response.data);  
    const photos= await fetch(url).then(res=>res.json())  
    setPhotos(photos.slice(0,20));  
  };  
  
  //Fetcha i dati solo al primo render  
  useEffect(() => {  
    getData();  
  }, []);  
  return (  
    <div>  
      {photos.map(photo => (  
        <img alt={photo.title} src={photo.url} />  
      ))}  
    </div>  
  );  
};
```

# Json Server

Vediamo ora come lavorare su dati presenti sul server in modo tale che la nostra applicazione interagisca con un backend.

Usiamo json server che simula un vero e proprio server. Se creiamo un tag nel json che si chiama post ci espone questi metodi

## Routes

Based on the previous `db.json` file, here are all the default routes. You can also add [other routes](#) using `--routes`.

### Plural routes

```
GET  /posts
GET  /posts/1
POST /posts
PUT  /posts/1
PATCH /posts/1
DELETE /posts/1
```

### Singular routes

```
GET  /profile
POST /profile
PUT  /profile
PATCH /profile
```

Si installa un package gli diamo la struttura che vogliamo avere dei nostri dati e da quel momento simula un db su cui possiamo eseguire post e get.

<https://www.npmjs.com/package/json-server>

Dalla console bash lanciamo ***npm i -g json-server*** e installiamolo globalmente. Nel frattempo trasformiamo la nostra struttura dati in clienteService in un json

```
const clienti=[...];
JSON.stringify(clienti);
```

Dopo di che creiamo un db.json nella root della nostra applicazione, incolliamo il json e mettiamo in ascolto il server su questo file eseguendo la seguente istruzione

```
json-server --watch db.json
```

Installiamo una nuova estensione di VC thunder client per testare le API Rest e configuriamo le nostre chiamate rest

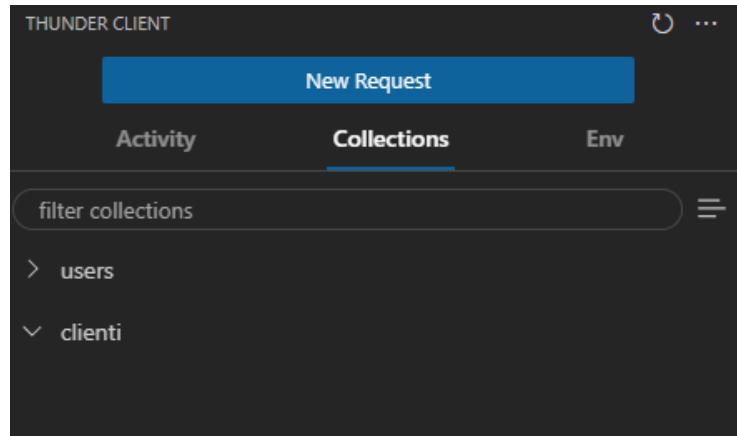
```
[{"id":1,"nome":"Rob1","cognome":"Del","codicefiscale":"DLSRRT77h29a132Q","email":"del1@del.it","telefono":"3288899222","note":"Nota 1"}, {"id":2,"nome":"Rob2","cognome":"Del","codicefiscale":"DLSRRT77h29a132Q","email":"del2@del.it","telefono":"3288899222","note":"Nota 2"}, {"id":3,"nome":"Rob3","cognome":"Del","codicefiscale":"DLSRRT77h29a132Q","email":"del3@del.it","telefono":"3288899222","note":"Nota 3"}, {"id":4,"nome":"Rob4","cognome":"Del","codicefiscale":"DLSRRT77h29a132Q","email":"del4@del.it","telefono":"3288899222","note":"Nota 4"}]
```



A screenshot of a code editor displaying a JSON object named "clienti". The object contains four client entries, each represented by a separate object with properties: id, nome, cognome, codicefiscale, email, telefono, and note. The "id" property is highlighted in yellow.

```
{
  "clienti": [
    {
      "id": 1,
      "nome": "Rob1",
      "cognome": "Del",
      "codicefiscale": "DLSRRT77h29a132Q",
      "email": "del1@del.it",
      "telefono": "3288899222",
      "note": "Nota 1"
    },
    {
      "id": 2,
      "nome": "Rob2",
      "cognome": "Del",
      "codicefiscale": "DLSRRT77h29a132Q",
      "email": "del2@del.it",
      "telefono": "3288899222",
      "note": "Nota 2"
    },
    {
      "id": 3,
      "nome": "Rob3",
      "cognome": "Del",
      "codicefiscale": "DLSRRT77h29a132Q",
      "email": "del3@del.it",
      "telefono": "3288899222",
      "note": "Nota 3"
    },
    {
      "id": 4,
      "nome": "Rob4",
      "cognome": "Del",
      "codicefiscale": "DLSRRT77h29a132Q",
      "email": "del4@del.it",
      "telefono": "3288899222",
      "note": "Nota 4"
    }
  ]
}
```

Creiamo una nuova collection clienti e poi le request per creare il post, get, put e delete.



POST  Send

Query Headers  Auth Body  Tests Pre Run New

Json Xml Text Form Form-encode Graphql Binary

Json Content Format

```
1 {  
2   "nome": "Rob Nuovo",  
3   "cognome": "Del Nuovo",  
4   "codicefiscale": "DLSRRT77h29a132Q",  
5   "email": "del_nuovo@del.it",  
6   "telefono": "3288899222",  
7   "note": "Nota 1"  
8 }  
9 }
```

GET  Send

Query Headers  Auth Body Tests Pre Run New

Http Headers

Accept \*/\*

User-Agent Thunder Client (<https://www.thunderclient.com>)

Accept application/json

header value

GET  Send

Query Headers  Auth Body Tests Pre Run New

Query Parameters

parameter value

PUT  Send

Query Headers  Auth Body  Tests Pre Run New

Json Xml Text Form Form-encode Graphql Binary

Json Content Format

```
1 {  
2   "nome": "Rob Update",  
3   "cognome": "Del Update",  
4   "codicefiscale": "DLSRRT77h29a132Q",  
5   "email": "del_Update@del.it",  
6   "telefono": "3288899222",  
7   "note": "Nota 1"  
8 }  
9 }
```

DELETE  Send

Query Headers  Auth Body Tests Pre Run New

# Esercizio 1

Riprendere l'esercizio sulla biblioteca e modificarlo per usare lo useState e lo useEffect.

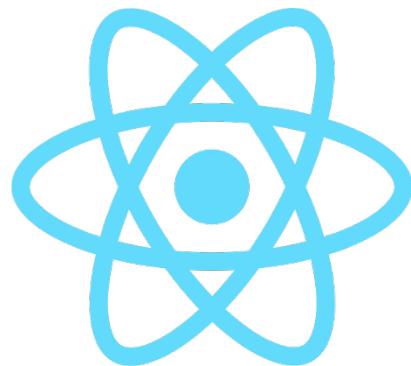
Usa json-server per creare un db di libri e le chiamate rest.

# Esercizio 2

Realizzare un componente che gestisce un cronometro. Impostare una variabile a zero e mostrarla a video. Aggiungere 3 bottoni : Start Stop e Reset.



# React



React

# Render Condizionale

Ora, parliamo del rendering condizionale. Questo permette di visualizzare diversi componenti a seconda delle condizioni specificate. Per esempio, potremmo decidere di visualizzare un componente o un altro a seconda dello stato di un'applicazione. Per esempio creiamo un componente RenderCondizionale e ritorniamo cosa visualizzare in relazione allo stato della variabile useState

```
import React, { useState } from "react";
💡
const RenderCondizionale = () => {
  const [loading, setLoading] = useState(false);

  if (loading) {
    return <h2>Loading...</h2>;
  }

  return (
    <div>RenderCondizionale</div>
  )
}

export default RenderCondizionale
```

# Render Condizionale

Quanto appena visto può essere sicuramente utile quando dobbiamo fetchare dei dati così da mostrare loading il tempo necessario al caricamento. Vediamo come fare.

Riprendiamo quanto visto prima per il fetching. Aggiungiamo anche uno useState isError.

Ovviamente se error sarà true manderemo a schermo il messaggio di errore.

I due return per loading ed error possiamo farli come due componenti.

```
const url = "https://jsonplaceholder.typicode.com/users";
const RenderCondizionale = () => {

  const [isLoading, setIsLoading] = useState(true);
  const [isError, setIsError] = useState(false);
  const [users, setUsers] = useState([]);

  const getData = async () => {
    setIsError(false);
    setIsLoading(true);
    try {
      const response = await axios.get(url);
      setUsers(response.data);
    } catch (error) {
      console.log(error);
      setIsError(true);
    }
    setIsLoading(false);
  };
}

if (isLoading) {
  return <Loading></Loading>;
}
if (isError) {
  return <ErrorRender></ErrorRender>;
}

return (
  <div>RenderCondizionale</div>
)
const Loading = () => {
  return <h2>Loading...</h2>;
}
const ErrorRender = () => {
  return <h2>Attenzione è avvenuto un errore</h2>;
}
```

# Render Condizionale

Come abbiamo visto un eventuale errore verrà intercettato dal try catch. Ora andiamo ad aggiungere lo useEffect e facciamo stampare gli utenti

```
useEffect(()=>{
    setTimeout(()=>{ //aggiunto per vedere chiaramente il cambio di componente
        getData();
    },3000)

},[])
if (isLoading) {
    return <Loading></Loading>;
}
if (isError) {
    return <ErrorRender></ErrorRender>;
}

return (
    <div><ul className="users">
    {users.map((el) => {
        const { name, id, username, email } = el;
        return (
            <li key={id} className="shadow">
                <div>
                    <h5>{id} - {name} - {username} - {email}</h5>
                </div>
            </li>
        );
    })}
    </ul></div>
)
```

# Nascondere Componente

Andiamo a vedere come possiamo nascondere e visualizzare un componente a seconda dello state di una variabile. Ad esempio se abbiamo una situazione del genere :

```
const Hidden = () => {  
  const [show, setShow] = useState(false);  
  
  return (  
    <div>  
      |   <h1>Nascondo</h1>  
    </div>  
  )  
}
```

Possiamo ora andare a creare un nuovo componente che utilizzeremo all'interno del componente appena creato in questo modo :

```
const Hidden = () => {
  const [show, setShow] = useState(false);

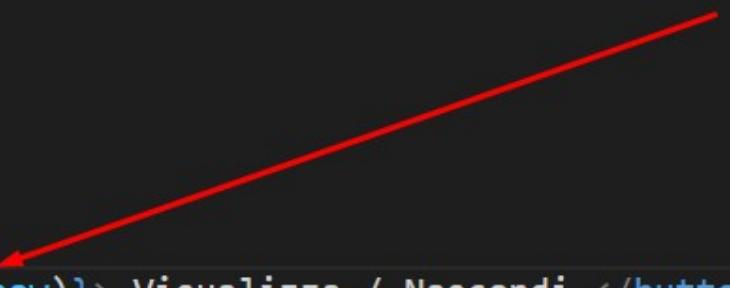
  return (
    <div>
      <h1>Nascondo</h1>
      <Elemento/>
    </div>
  )
}

const Elemento = () => {
  return (
    <div>
      <h2>Componente Elemento</h2>
    </div>
  )
}

export default Hidden
```

Nel componente principale andiamo ora a creare un bottone che al verificarsi dell'evento onClick andrà a settare il valore di show a false se era vero, e vero se era falso, in questo modo :

```
const Hidden = () => {  
  const [show, setShow] = useState(false);  
  
  return (  
    <div>  
      <h1>Nascondo</h1>  
      <button onClick={() => setShow(!show)}> Visualizza / Nascondi </button>  
      <Elemento/>  
    </div>  
  )  
}
```



# Operatore Ternario

Andiamo ora a vedere come sia possibile «cambiare» la visualizzazione di un componente in base al valore di una variabile.

Ci viene naturale pensare che per risolvere il problema, potremmo utilizzare un classico IF, ma ricordiamoci che all'interno del rendering di un componente stiamo utilizzando **JSX** e **questo linguaggio non ci consente di utilizzare costrutti condizionali.**

Quindi ci viene in aiuto **l'operatore ternario**, che è una scorciatoia di un costrutto if che a differenza di una if **ritorna sempre un valore**. Questa è la sintassi :

```
nomeDaVerificare === 'valore' ? 'sono vero' : 'sono falso'
```

Se invece la variabile è un booleano basta mettere il nome della variabile :

```
nomeDaVerificare ? 'sono vero' : 'sono falso'
```

Per riprendere il nostro esempio possiamo modificare il bottone che cambia il suo valore in base al valore della variabile show in questo modo :

```
return (
  <div>
    <h1>Nascondo</h1>
    <button onClick={() => setShow(!show)}> {
      show ? 'Nascondi' : 'Visualizza'
    } </button>
    <Elemento/>
  </div>
)
```



# Short Circuit Evaluation

Una short circuit evaluation è un'espressione che torna un valore in base allo stato di una costante o di una variabile che vogliamo andare a valutare. Esistono 2 tipi di short circuit :

**And &&**

**Or ||**

||

Esegue il controllo dell'espressione, se è vuota prende il secondo valore, **altrimenti** il primo :

const esempio = parola **||** "paperino"

Se la variabile parola ha un valore, allora *esempio* assume il valore di parola, altrimenti assume il valore "paperino".

&&

Esegue il controllo dell'espressione, se è vera prende il secondo valore :

const esempio = parola **&&** "paperino"

Se la variabile parola ha un valore, allora esempio assume il valore di "paperino".

Quindi ritornando al nostro esempio posso utilizzare lo Short Circuit && per visualizzare o meno il componente Elemento in base al valore di show :

```
const Hidden = () => {  
  
  const [show, setShow] = useState(false);  
  
  return (  
    <div>  
      <h1>Nascondo</h1>  
      <button onClick={() => setShow(!show)}> {  
        show ? 'Nascondi' : 'Visualizza'  
      } </button>  
  
      {show && <Elemento/>} 

React


```

Visto come gestire la visualizzazione condizionata degli elementi torniamo un attimo sullo useEffect. Mettiamo un contatore sul componente elemento che aumenta ogni secondo:

```
const Elemento = () => {
  const [contatore, setContatore] = useState(0);

  useEffect(() => {
    const timer = setTimeout(() => {
      setContatore((oldValue) => oldValue + 1);
    }, 1000);
    return () =>
  }, [contatore]);
  return (
    <div>
      <h2>{contatore}</h2>
    </div>
  );
}
```

Se non mettiamo la funzione di pulizia del contatore sul render del componente avremo questo messaggio in console:

```
[HMR] Waiting for update signal from WDS... log.js:24
✖ Warning: Can't perform a React state update on an unmounted component. This is a no-op, but it indicates a memory leak in your application. To fix, cancel all subscriptions and asynchronous tasks in a useEffect cleanup function.
    at Elemento (http://localhost:3000/static/js/main.chunk.js:420:91)
>
```

```
useEffect(() => {
  const timer = setTimeout(() => [
    setContatore(oldValue) => oldValue + 1),
  ], 1000);
  return () => clearTimeout(timer);
}, [contatore]);
```

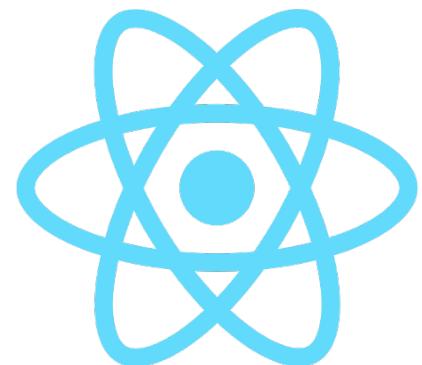
# Esercizio

Creare un componente con un elenco di News composte da id, titolo e descrizione. Salvare in un file esterno 5 news con testo a piacimento ed all'interno del componente elencarle. Aggiungere un bottone finale che cambia il colore del font e dello sfondo di tutta la pagina. Situazione iniziale sfondo bianco scritte nere. Se clicco sul bottone sfondo nero e scritte bianche. Ad ogni click invertire nuovamente i colori.

Aggiungere bottone Nascondi / Visualizza News

- 1 – Creare File Esterno con le news
- 2 – Creare Componente Card che prende i dati dei prodotti (no img)
- 3 – Elencare i prodotti
- 4 – Aggiungere pulsante che cambia il colore delle card
- 5 – Aggiungere pulsante che cambia il colore dello sfondo della pagina

# React



# Form

Vediamo in questa lezione la gestione dei form con React partendo dal form di esempio messo a disposizione di bootstrap tenendo presente due regole :

- Sostituire class con **className**
- Sostituire il for della label con **htmlFor**

```
const Form = () => {
  return (
    <div className="container">
      <form className="row g-3">
        <div className="col-md-6">
          <label htmlFor="inputNome" className="form-label">Nome</label>
          <input type="text" className="form-control" id="inputNome" />
        </div>
        <div className="col-md-6">
          <label htmlFor="inputCognome" className="form-label">Cognome</label>
          <input type="text" className="form-control" id="inputCognome" />
        </div>
        <div className="col-12">
          <button type="submit" className="btn btn-primary">Invia</button>
        </div>
      </form>
    </div>
  )
}
```

```
const Form = () => {
  return (
    <div className="container">
      <form className="row g-3">
        <div className="col-md-6">
          <label htmlFor="inputNome" className="form-label">Nome</label>
          <input type="text" className="form-control" id="inputNome" />
        </div>
        <div className="col-md-6">
          <label htmlFor="inputCognome" className="form-
label">Cognome</label>
          <input type="text" className="form-control" id="inputCognome" />
        </div>
        <div className="col-12">
          <button type="submit" className="btn btn-primary">Invia</button>
        </div>
      </form>
    </div>
  )
}

React
```

# onSubmit

L'evento che «intercetta» il submit del form è **onSubmit** che serve per «gestire» l'invio del form e va applicato al tag <form> in questo modo :

```
const gestioneDati = () => {
  console.log('gestione dati di un form');
}

return (
  <div className="container">
    <form className="row g-3" onSubmit={gestioneDati}>
      <div className="col-md-6">
```

# event.preventDefault

Se proviamo ora ad inserire un nome ed un cognome e clicchiamo su invia, cosa succede ? Succede che React intercetta l'evento onSubmit, quindi esegue il la funzione corrispondente ed alla fine dell'esecuzione della arrow function procede con **'l'invio del form'** che si traduce in un caricamento della pagina.

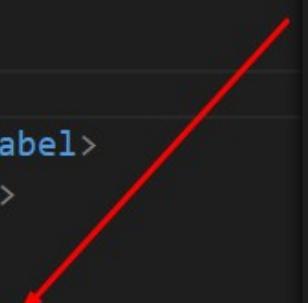
Per «prevenire» o meglio, per impedire che l'evento onSubmit completa il suo lavoro, possiamo intervenire passando alla funzione onSubmit l'evento e richiamare al suo interno la funzione preventDefault. Con questa funzione stiamo dicendo al compilatore di non effettuare il submit, ma che verrà gestito manualmente dal programmatore.

```
const Form = () => {  
  const gestioneDati = (e) => {  
    e.preventDefault();  
    console.log('gestione dati di un form');  
  }  
  return (  
    <div className="container">  
      <form className="row g-3" onSubmit={gestioneDati}>  
        <div className="col-md-6">
```

Se proviamo ora il nostro codice possiamo verificare che chiama comunque la funzione di submit ma non effettua il submit, rimanendo sulla pagina in attesa di un submit «manuale».

Questa gestione del submit posso farla attraverso l'evento onSubmit applicato al form (come abbiamo fatto in questo esempio) oppure attraverso l'evento onClick applicato al bottone, è la stessa cosa!

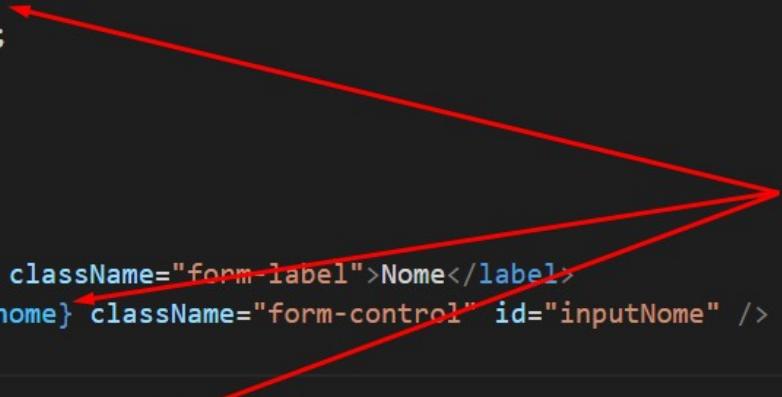
```
<form className="row g-3" >
  <div className="col-md-6">
    <label htmlFor="inputNome" className="form-label">Nome</label>
    <input type="text" className="form-control" id="inputNome" />
  </div>
  <div className="col-md-6">
    <label htmlFor="inputCognome" className="form-label">Cognome</label>
    <input type="text" className="form-control" id="inputCognome" />
  </div>
  <div className="col-12">
    <button type="submit" className="btn btn-primary" onClick={gestioneDati}>Invia
  </div>
</form>
```



Ora andiamo a dare un valore al nostro input e lo collegiamo ad una variabile dichiarata utilizzando useState in questo modo :

```
const [nome, setNome] = useState('');
const [cognome, setCognome] = useState('');

return (
  <div className="container">
    <form className="row g-3" >
      <div className="col-md-6">
        <label htmlFor="inputNome" className="form-label">Nome</label>
        <input type="text" value={nome} className="form-control" id="inputNome" />
      </div>
      <div className="col-md-6">
        <label htmlFor="inputCognome" className="form-label">Cognome</label>
        <input type="text" value={cognome} className="form-control" id="inputCognome" />
      </div>
      <div className="col-12">
        <button type="submit" className="btn btn-primary" onClick={gestioneDati}>Invia</button>
      </div>
    </form>
  </div>
)
```



The diagram consists of two red arrows originating from the useState declarations in the first two lines of the code. One arrow points to the 'value' prop of the first input field, and the other points to the 'value' prop of the second input field.

Così facendo possiamo notare che ora non posso scrivere nulla all'interno dell'input in quanto ho passato il «controllo» alla funzione useState.

Per ovviare a questo problema devo aggiungere nel nostro input l'evento **onChange** passando in ingresso l'evento e tramite una arrow function passare alla funzione set della useState l'evento target.value (target è l'evento del click sull'input) in questo modo :

```
<form className="row g-3" >
  <div className="col-md-6">
    <label htmlFor="inputNome" className="form-label">Nome</label>
    <input type="text" value={nome} onChange={(e) => setNome(e.target.value)} className="form-control" id="inputNome" />
  </div>
  <div className="col-md-6">
    <label htmlFor="inputCognome" className="form-label">Cognome</label>
    <input type="text" value={cognome} onChange={(e) => setCognome(e.target.value)} className="form-control" id="inputCognome" />
  </div>
  <div className="col-12">
    <button type="submit" className="btn btn-primary" onClick={gestioneDati}>Invia</button>
  </div>
</form>
```

In questo modo possiamo interagire con il nostro input ed andare ad inserire dei controlli sui dati inseriti. Possiamo anche creare una funzione handler da chiamare sull'onChange

```
const handleChange = (e) => {
  const { value } = e.target;
  setCellulare(value);
};
```

React

```
      <input type="text" value={cellulare} onChange={handleChange} className="form-control col-9" ></input>
```

# Submit

Per vedere come è possibile gestire un submit aggiungiamo all'interno del nostro componente il seguente codice. Vediamo che se clicchiamo sul pulsante effettivamente abbiamo il log nella console e il form viene svuotato

```
const gestioneDati = (e) => {
  e.preventDefault();

  if (nome && cognome) {
    console.log(nome,cognome)
    setNome("");
    setCognome("");
  } else {
    alert("riempi il form");
  }

};
```

Ora aggiungiamo all'interno del nostro componente un array di persone ed ogni volta che clicchiamo su invio viene aggiunta una persona a questo array :

```
const [nome, setNome] = useState('');
const [cognome, setCognome] = useState('');
const [persone, setPersone] = useState([]);  
return (
```



A questo punto utilizzando il metodo gestioneDati eseguito al momento del click del form possiamo popolare il nostro array come già sappiamo fare utilizzando lo spread operator :

```
const gestioneDati=(e)=>{
  e.preventDefault();
  if (nome && cognome) {
    setPersone([
      ...persone,
      {
        nome,
        cognome,
      },
    ]);
    setNome("");
    setCognome("");
  } else {
    alert("riempi il form");
  }
}


```

Ora modifichiamo l'esempio usando un oggetto piuttosto che singole variabili. Aggiungiamo quindi l'oggetto persona, modifichiamo il value dei campi del form e creiamo una funzione handleChange per gestire la modifica del valore

```
const [persona, setPersona] = useState({
  nome: "",
  cognome: "",
});
```

```
<div className="col-md-6">
  <label htmlFor="inputNome" className="form-label">Nome</label>
  <input type="text" value={persona.nome} onChange={handleChange} className="form-control" />
</div>
<div className="col-md-6">
  <label htmlFor="inputCognome" className="form-label">Cognome</label>
  <input type="text" value={persona.cognome} onChange={handleChange} className="form-control" />
</div>
<div className="col-md-12">
```

```
const handleChange = (e) => {
  const { name, value } = e.target;
  setPersona({ ...persona, [name]: value });
};
```

# Andiamo quindi a popolare il nostro array persone in questo modo

```
const gestioneDati=(e)=>{  
  
    e.preventDefault();  
  
    if (persona.nome && persona.cognome) {  
        setPersone([  
            ...persone,  
            {  
                ...persona,  
            },  
        ]);  
        setPersona({  
            nome: "",  
            cognome: ""  
        });  
    } else {  
        alert("riempi il form");  
    }  
}
```

# useRef

Con useRef, possiamo ottenere un riferimento diretto a un elemento DOM all'interno dei nostri componenti React. Ad esempio, immaginate di avere una serie di paragrafi (`<p>`) e alla fine un bottone che, quando premuto, scorrerà automaticamente fino all'ultimo paragrafo della pagina.

```
const RefExample = () => {
  const ref = React.useRef(null);
  console.log(ref);
```

# useRef

Associamo il ref al paragrafo e creiamo un pulsante con la funzione di scroll sull'onclick

```
<p>Questo è un paragrafo.</p>
<p>Questo è un paragrafo.</p>
<p>Questo è un paragrafo.</p>
<p>Questo è un paragrafo.</p>
<p ref={ref}>Questo è l'ultimo paragrafo.</p>
<div
  style={{
    height: "30vh",
  }}
></div>
```

```
const handleScroll = () => {
  if (!ref || !ref.current) {
    return;
  }

  ref.current.scrollIntoView({ behavior: "smooth", block: "center" });
};

return [
  <>
    <h1>Use Ref</h1>
    <div
      className="my-5 mx-auto"
      style={{
        height: "100vh",
      }}
    >
      <button className="btn btn-info" onClick={handleScroll}>
        Scroll
      </button>
    </div>
  </>
]
```

# useRef

useRef può essere usato anche per riferirci a campi di input. Il precedente esempio sui form potrebbe dunque seguire

```
1 import React, { useRef, useState } from 'react'
2
3 const FormUseRef = () => {
4   const nomeRef=useRef(null);
5   const cognomeRef=useRef(null);
6
7   const [persone, setPersone]=useState([]);
8
9   const handlerSubmit=(e)=> {
10     e.preventDefault();
11     const nome = nomeRef.current.value;
12     const cognome = cognomeRef.current.value;
13     if(nome && cognome){
14
15       setPersone([
16         ...persone,
17         { nome, cognome }
18       ])
19       nomeRef.current.value = '';
20       cognomeRef.current.value = '';
21
22     }else{
23       console.log("compilare nome e cognome")
24     }
25
26     console.log("Submit form");
27   }
28 }
```

```
return (
  <div className="container">
    <h1>UseREF</h1>
    <form className="row g-3" onSubmit={handlerSubmit}>
      <div className="col-md-6">
        <label htmlFor="inputNome" className="form-label">Nome</label>
        <input type="text" ref={nomeRef} name="nome" className="form-control" />
      </div>
      <div className="col-md-6">
        <label htmlFor="inputCognome" className="form-label">Cognome</label>
        <input type="text" ref={cognomeRef} name="cognome" className="form-control" />
      </div>
      <div className="col-md-12">
        <button type="submit" className="btn btn-primary">Invia</button>
      </div>
    </form>
    <div>
      {
        persone.map((el,index)=>{
          const {nome,cognome}=el
          return(<p key={index}>{nome} {cognome}</p>)
        })
      }
    </div>
  </div>
)
```

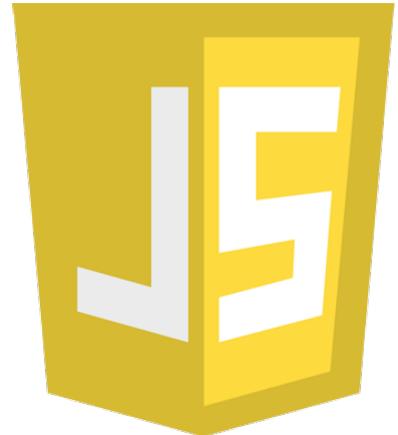
# Esercizio

Creare un nuovo progetto con un componente che visualizza un menu con le seguenti voci : Home – Chi Sono – Contatti.

Al click di ogni singola voce viene visualizzato un componente. Sulla home inserire un'immagine di presentazione a piacere. Su chi sono un breve testo che vi presenta e su contatti un form compilabile che all'invio visualizza un messaggio di conferma solo se sono stati compilati tutti i campi (i campi sono nome, cognome, email, cellulare e messaggio).

# JAVASCRIPT

Gli oggetti



# Oggetto

L'oggetto è una variabile. Una variabile specifica più complessa di quelle viste finora. Ogni oggetto è composto da una o più proprietà, ed ogni proprietà ha il suo valore. Ad esempio il modo per creare un semplice oggetto è :

```
var automobile = {marca :" Fiat", modello :  
"500", colore: " Rosso"}
```

# Prime osservazioni

Possiamo subito notare che un oggetto non è altro che una variabile con delle proprietà ognuna separata da una virgola.

E ad ogni proprietà viene assegnato un valore tramite i due punti : il tutto racchiuso da parentesi graffe e terminate con il solito punto e virgola.

Per accedere ad una proprietà di un oggetto è sufficiente richiamare il nome della variabile e tramite l'operatore dot (**il punto**) posso «entrare» nell'oggetto e richiamare la proprietà. Nel nostro esempio se voglio stampare a video la marca dell'oggetto automobile posso scrivere:

```
document.write(automobile.marca);
```

Un altro modo alternativo per dichiarare ed istanziare un oggetto è nel seguente modo :

```
var automobile = new Object();
automobile.marca = "Fiat";
automobile.modello = "500";
automobile.colore = "Rosso";
```

```
Document.write(automobile.marca);
```

# Oggetti Innestati

E' possibile avere l'esigenza di inserire un oggetto all'interno di un altro oggetto, in altre parole di innestare un oggetto all'interno di un altro.  
E' possibile farlo tramite la seguente sintassi :

```
var persona = {  
    nome: 'Mario',  
    cognome: 'Rossi',  
    eta: 30,  
    indirizzo: {  
        via: 'Via casilina',  
        civico: 23  
    }  
}
```

Per richiamare una proprietà di un oggetto innestato è sufficiente usare sempre l'operatore dot partendo sempre dalla classe principale :

```
document.write(persona.indirizzo.via);
```

E se invece di un indirizzo voglio dare la possibilità di inserire due o più indirizzi? Posso usare un array come già sappiamo :

```
var persona = {  
    nome: "Mario",  
    cognome: "Rossi",  
    eta: 30,  
    indirizzo: [{  
        via: "Via casilina",  
        civico: 23  
    }, {  
        via: "Via prenestina",  
        civico: 110  
    }]  
}  
  
document.write(persona.indirizzo[1].via);
```

# Funzioni nell'oggetto

Naturalmente come ogni linguaggio di programmazione, gli oggetti, oltre ad avere le proprietà, hanno anche le funzioni.

Per scrivere una funzione all'interno di un oggetto è sufficiente dare un nome alla funzione seguito dai : e dalla parola chiave function seguita dalle parentesi tonde () :

```
var persona = {
    nome: 'Mario',
    cognome: 'Rossi',
    eta: 30,
    indirizzo: [
        {
            via: 'Via casilina',
            civico: 23
        },
        {
            via: "Via prenestina",
            civico: 110
        }
    ],
    stampa: function(nome,cognome){
        document.write(nome + ' ' + cognome);
    }
}

persona.stampa("Riccardo","Cattaneo");
```

# Array Associativi

Vediamo un altro esempio di oggetto.

```
let book = {  
    "main title": "JavaScript",  
    "sub-title": "The Definitive Guide",  
    for: "all audiences",  
    author: {  
        firstname: "David",  
        surname: "Flanagan"  
    }  
};
```

```
let book = {
    "main title": "Libro Javascript",
    "sub-title": "Linguaggio potente",
    for: "Developer",
    author: {
        firstname: "Rob",
        lastname: "Del"
    }
}
```

Possiamo notare che alcuni attributo dell'oggetto sono con spazi e tra apici. In questo caso per richiamarli dobbiamo usare la seguente istruzione

**book**["main title"]

# Array Associativi

Per spiegare il caso precedente facciamo un ulteriore esempio

```
object.property  
object["property"]
```

La prima sintassi, che utilizza il punto e un identificatore, è simile alla sintassi utilizzata per accedere a un campo statico di una struct o di un oggetto in C o Java.

La seconda sintassi, che usa le parentesi quadre e una stringa, assomiglia all'accesso a un array, ma a un array indicizzato da stringhe anziché da numeri. Questo tipo di array è noto come array associativo (o hash o mappa o dizionario).

Gli oggetti JavaScript sono array associativi.

# Modello a oggetti

JavaScript è un linguaggio ad oggetti basato su **prototipi**, piuttosto che sulle classi. A causa di questa diversa base, può essere meno evidente come JavaScript permette di creare gerarchie di oggetti e di avere l'ereditarietà delle proprietà e dei loro valori.

I linguaggi ad oggetti basati su classi, come **Java**, si basano su due entità distinte: le **classi** e le **istanze**.

**Una classe** definisce tutte le proprietà che caratterizzano una determinata collezione di oggetti. Una classe è un'entità astratta, più che un membro del gruppo di oggetti che descrive. Per esempio, la classe `Impiegato` può rappresentare il gruppo di tutti i dipendenti.

**Un'istanza**, d'altra parte, è l'istanziazione di una classe; cioè uno dei suoi membri. Per esempio Luca può essere un'istanza della classe `Impiegato`, che rappresenta un particolare individuo come un dipendente. Un'istanza ha esattamente le stesse proprietà della classe a cui appartiene.

Un linguaggio basato su prototipi, come JavaScript, non fa questa distinzione: **ha solo oggetti**. Introduce la nozione di oggetto prototipo (prototypical object), un oggetto usato come modello da cui prendere le proprietà iniziali per un nuovo oggetto.

Ogni oggetto può specificare le sue proprietà, anche quando viene creato o in fase di esecuzione. Inoltre, ogni oggetto può essere associato ad un altro oggetto come prototipo, consentendo al secondo oggetto di condividere le proprietà del primo.

Nei linguaggi basati su classi, le classi vengono definite in classi separate. In queste definizioni è possibile specificare metodi speciali, chiamati costruttori, per creare istanze della classe. Un costruttore può specificare i valori iniziali per le proprietà dell'istanza ed eseguire altre elaborazioni adatte al momento della creazione. Per creare le istanze di una classe si utilizza l'operatore new associato al metodo costruttore.

JavaScript segue un modello simile, ma **non prevede la definizione della classe separata dal costruttore**. Invece, per creare oggetti con un particolare set di proprietà e valori si definisce una **funzione costruttore**. **Ogni funzione JavaScript può essere usata come costruttore**. Per creare un nuovo oggetto si utilizza l'operatore new associato a una funzione costruttore.

# Costruttore

In Javascript non c'è bisogno di definire una classe ma definiamo direttamente l'oggetto così come ci serve al momento ed eventualmente possiamo modificare la sua struttura nel corso dell'esecuzione del nostro script.

Immaginiamo però di aver bisogno di più oggetti dello stesso tipo, ad esempio di più oggetti persona, che condividono la stessa struttura:

```
var persona = {  
    nome: "Mario",  
    cognome: "Rossi",  
    indirizzo: "Via Garibaldi, 50 - Roma",  
    email: "mario.rossi@html.it",  
    mostraNomeCompleto: function() { ... },  
    calcolaCodiceFiscale: function() { ... }  
}
```

Utilizzando la **notazione letterale** saremmo costretti a ripetere la definizione per ciascun oggetto che vogliamo creare. In altre parole, ricorrendo alla notazione letterale nella definizione degli oggetti otteniamo un risultato non riutilizzabile.

Per evitare quindi di dover ridefinire da zero oggetti che hanno la stessa struttura possiamo ricorrere ad un costruttore. Un costruttore non è altro che una normale funzione JavaScript invocata mediante l'operatore new. Vediamo ad esempio come creare un costruttore per l'oggetto persona:

```
function persona() {  
    this.nome = "";  
    this.cognome = "";  
    this.indirizzo = "";  
    this.email = "";  
    this.mostraNomeCompleto = function() {...};  
    this.calcolaCodiceFiscale = function() {...};  
}
```

Questa funzione definisce le proprietà del nostro oggetto assegnandole a se stessa (**this**) in qualità di oggetto ed impostando i valori predefiniti. Per creare un oggetto di tipo persona dovremo a questo punto invocare la funzione premettendo l'operatore **new**:

```
var marioRossi = new persona();
marioRossi.nome = "Mario";
marioRossi.cognome = "Rossi";
```

```
var giuseppeVerdi = new persona();
giuseppeVerdi.nome = "Giuseppe";
giuseppeVerdi.cognome = "Verdi";
```

In questo modo nella creazione di più oggetti con la stessa struttura ci limiteremo ad impostare i soli valori specifici che differenziano un oggetto dall'altro.

Nella definizione di un costruttore possiamo prevedere la presenza di **parametri** che possiamo utilizzare nell'inizializzazione del nostro oggetto. Consideriamo ad esempio la seguente definizione del costruttore dell'oggetto persona:

```
function persona(nome, cognome) {  
    this.nome = nome;  
    this.cognome = cognome;  
    this.indirizzo = "";  
    this.email = "";  
    this.mostraNomeCompleto = function() {...};  
    this.calcolaCodiceFiscale = function() {...};  
}  
}
```

Esso ci consente di creare ed inizializzare un oggetto specificando i valori nella chiamata al costruttore:

```
var marioRossi = new persona("Mario", "Rossi");
var giuseppeVerdi = new persona("Giuseppe", "Verdi");
```

È fondamentale utilizzare l'operatore **new** nella creazione di un oggetto tramite costruttore. Infatti se lo omettiamo, magari per dimenticanza, quello che otterremo non sarà la creazione di un oggetto ma l'esecuzione della funzione, con risultati imprevedibili. Ad esempio, se tentiamo di creare un oggetto persona omettendo l'operatore new:

**var marioRossi = persona();**

il valore della variabile marioRossi sarà **undefined**, dal momento che la funzione persona() non restituisce alcun valore.

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.

# Prototype

La flessibilità degli oggetti JavaScript si esprime principalmente nella possibilità di **modificare la struttura anche dopo la creazione**. Anche utilizzando un costruttore per creare un oggetto, continuiamo a disporre di questa possibilità.

Riprendendo il costruttore creato nelle lezioni precedenti (persona), possiamo ad esempio scrivere il seguente codice:

```
var marioRossi = new persona("Mario", "Rossi");
var giuseppeVerdi = new persona("Giuseppe", "Verdi");

marioRossi.telefono = "0612345678";
```

Creando una nuova proprietà telefono per l'oggetto Mario Rossi, senza influenzare la struttura di Giuseppe Verdi. In sostanza, nella creazione di oggetti possiamo partire da una struttura comune definita da un costruttore per poi personalizzarla in base alle nostre esigenze.

Ma come fare per modificare la struttura di tutti gli oggetti creati tramite un costruttore? Se ad esempio, dopo aver creato diversi oggetti dal costruttore persona() vogliamo aggiungere per tutti la proprietà telefono, possiamo sfruttare una delle caratteristiche più interessanti della programmazione ad oggetti di JavaScript: il **prototype**. Nel nostro caso procederemo nel seguente modo:

**persona.prototype.telefono = "123456";**

Questo assegnamento fa sì che **tutti gli oggetti creati tramite il costruttore persona()** abbiano istantaneamente tra le loro proprietà anche la proprietà telefono valorizzata al valore indicato.

Ad essere precisi, la nuova proprietà non è direttamente agganciata a ciascun oggetto, ma accessibile come se fosse una sua proprietà. Questo grazie al meccanismo di prototyping che sta alla base dell'ereditarietà nella programmazione ad oggetti in JavaScript.

# Metodi apply() e call()

JavaScript, come già sappiamo, considera tutto come un oggetto e quindi rappresenta in questo modo anche le funzioni (e quindi i metodi). Ciascuna funzione presenta infatti proprietà e metodi proprio come un normale oggetto. Il prototipo dell'oggetto Function presenta due metodi di fondamentale importanza che sono **apply()** e **call()**.

Il loro comportamento è simile, presentano una semplice differenza nella gestione dei parametri. Essi permettono infatti di invocare una determinata funzione definendo quale sarà il suo scope passandoglielo come primo parametro. È possibile inoltre fornire ulteriori parametri che verranno in qualche modo “passati” alla funzione oggetto dell’invocazione.

La differenza tra apply e call è proprio questa: apply() oltre all'oggetto scope accetta un **array di parametri**, mentre call() accetta un **numero indefinito di parametri**. Passiamo all'esempio:

```
function myFunction(a, b) {  
    return a * b;  
}  
myObject = myFunction.call(myObject, 10, 2);
```

```
function myFunction(a, b) {  
    return a * b;  
}  
myArray = [10, 2];  
myObject = myFunction.apply(myObject, myArray);
```

# Esercizio 3.1

Progettare l'oggetto canzoni che contiene delle proprietà: canzone1, canzone2, ecc... Queste proprietà a sua volta sono degli oggetti che hanno altre proprietà: titolo, nomeCantante e anno. Inserire poi alcuni dati a piacere e visualizzarli. Dopo fare inserire all'utente delle nuove canzoni attraverso il prompt dei comandi. Visualizzare il nuovo oggetto così costruito.

## Esercizio 3.2

Progettare un oggetto rubrica che ha come proprietà gli utenti. Dopo, per ogni utente specificare altre proprietà: utente, nome, cognome, telefono e indirizzo. Dove indirizzo è a sua volta un altro oggetto contenente altre proprietà.

Quindi popolare la rubrica con dei dati a piacere. Dopo eliminare l'ultimo elemento e visualizzare nuovamente la rubrica così ottenuta.

# Esercizio 3.3

1. Definisci un oggetto "studente" con le seguenti proprietà:
  - nome: una stringa con il nome dello studente
  - cognome: una stringa con il cognome dello studente
  - voto: un numero che rappresenta il voto dello studente
2. Aggiungi un metodo all'oggetto "studente" chiamato "calcolaMedia" che calcola la media di due voti passati come argomenti e restituisce il risultato.
3. Crea un nuovo oggetto "studente1" utilizzando l'oggetto "studente" come modello. Assegna valori alle proprietà "nome", "cognome" e "voto".
4. Chiama il metodo "calcolaMedia" sull'oggetto "studente1" passando due voti come argomenti. Stampa il risultato.
5. Crea un altro oggetto "studente2" utilizzando l'oggetto "studente" come modello. Assegna valori alle proprietà "nome", "cognome" e "voto".
6. Chiama il metodo "calcolaMedia" sull'oggetto "studente2" passando due voti come argomenti. Stampa il risultato.
7. Confronta i voti dei due studenti ("studente1" e "studente2") e stampa un messaggio che indica quale studente ha ottenuto un voto più alto.
8. Modifica il voto dello "studente1" e del "studente2" assegnando nuovi valori alla proprietà "voto".
9. Chiama nuovamente il metodo "calcolaMedia" sui due studenti con i nuovi voti. Stampa i nuovi risultati.
10. Confronta i nuovi voti dei due studenti e stampa un messaggio che indica quale studente ha ottenuto un voto più alto.