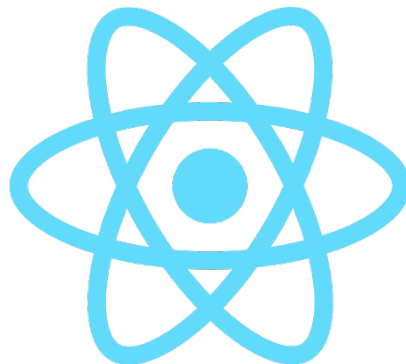


# React

## 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 corso 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 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.

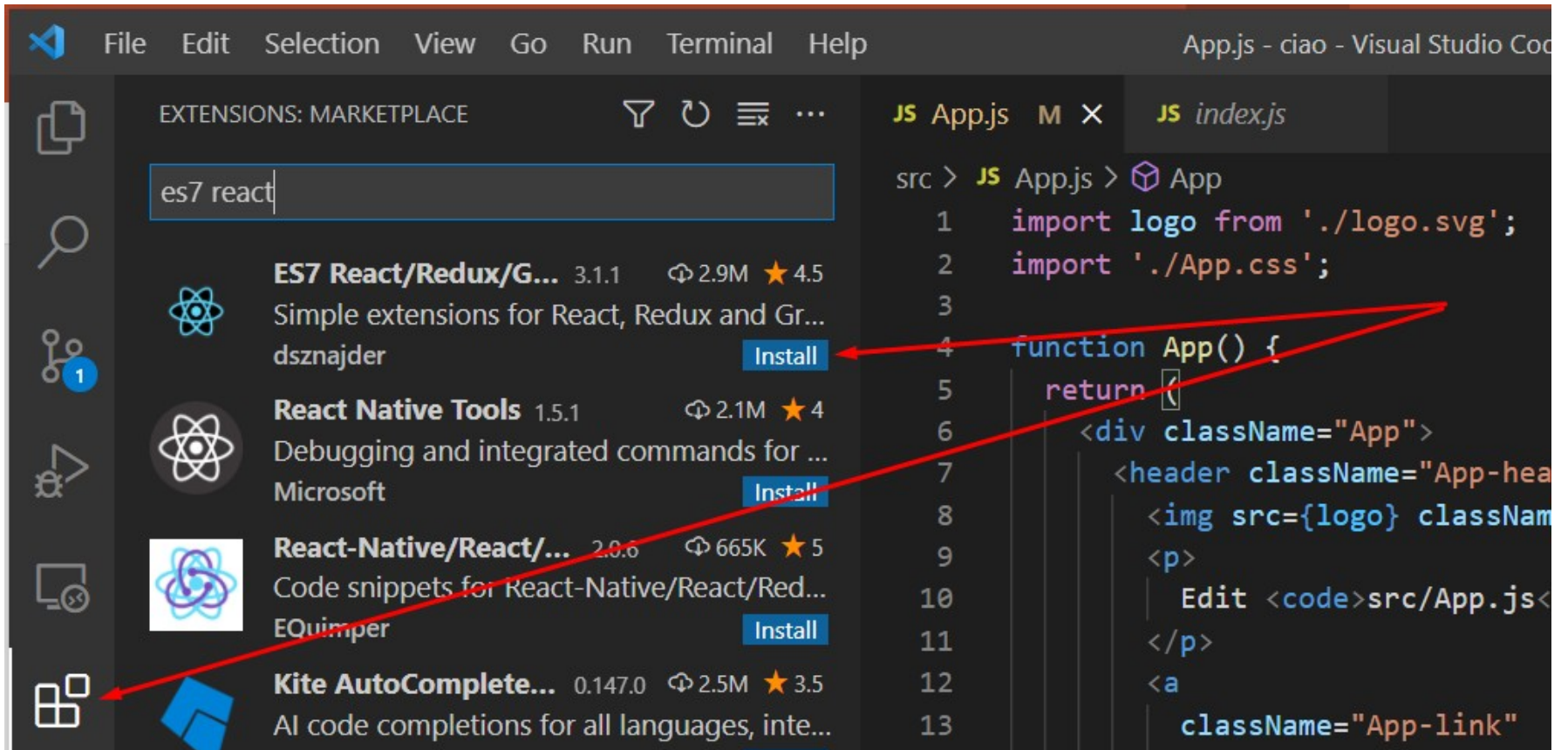
# Creare un Componente

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

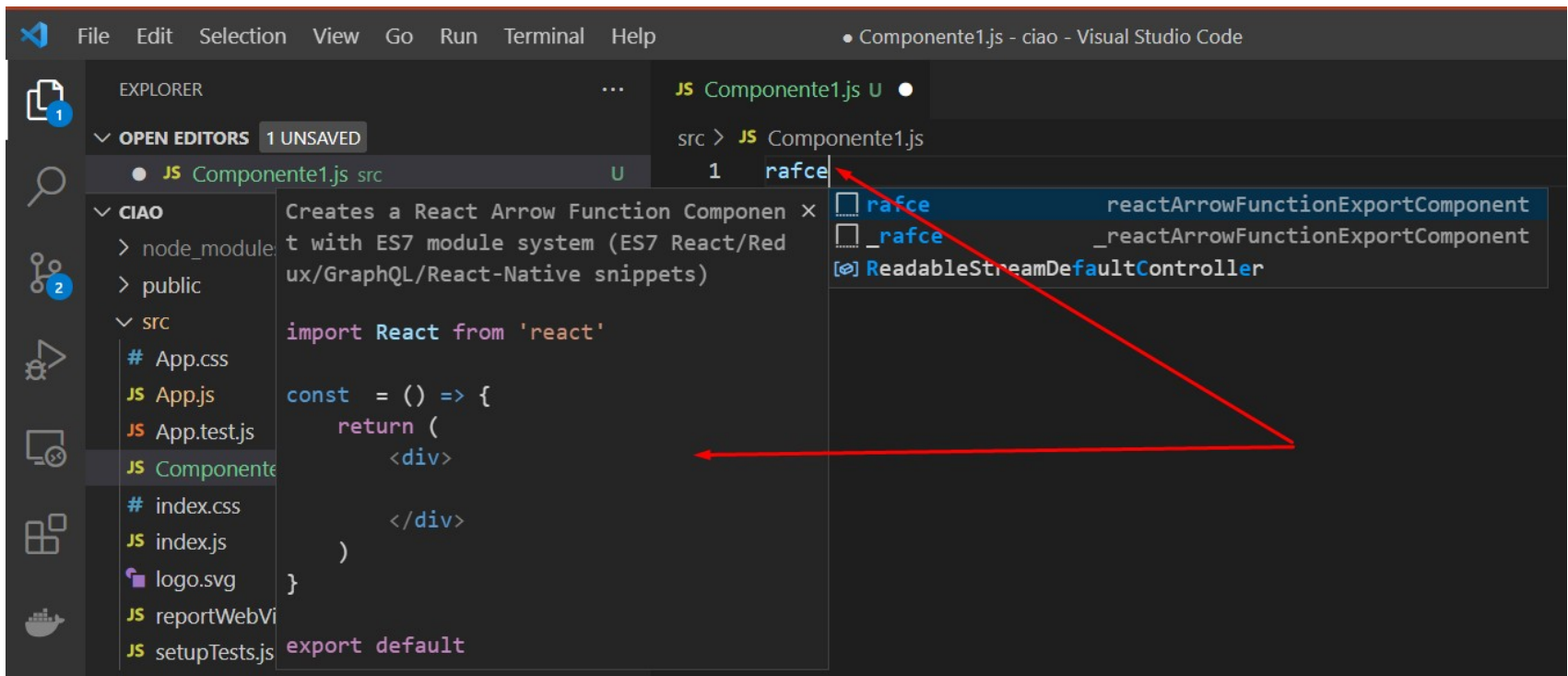
In definitiva, con React, creiamo elementi HTML personalizzati e li combiniamo per realizzare un'interfaccia utente. Ma basta con la teoria, è ora di immergerci nel codice e iniziare a costruire questi componenti.

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.



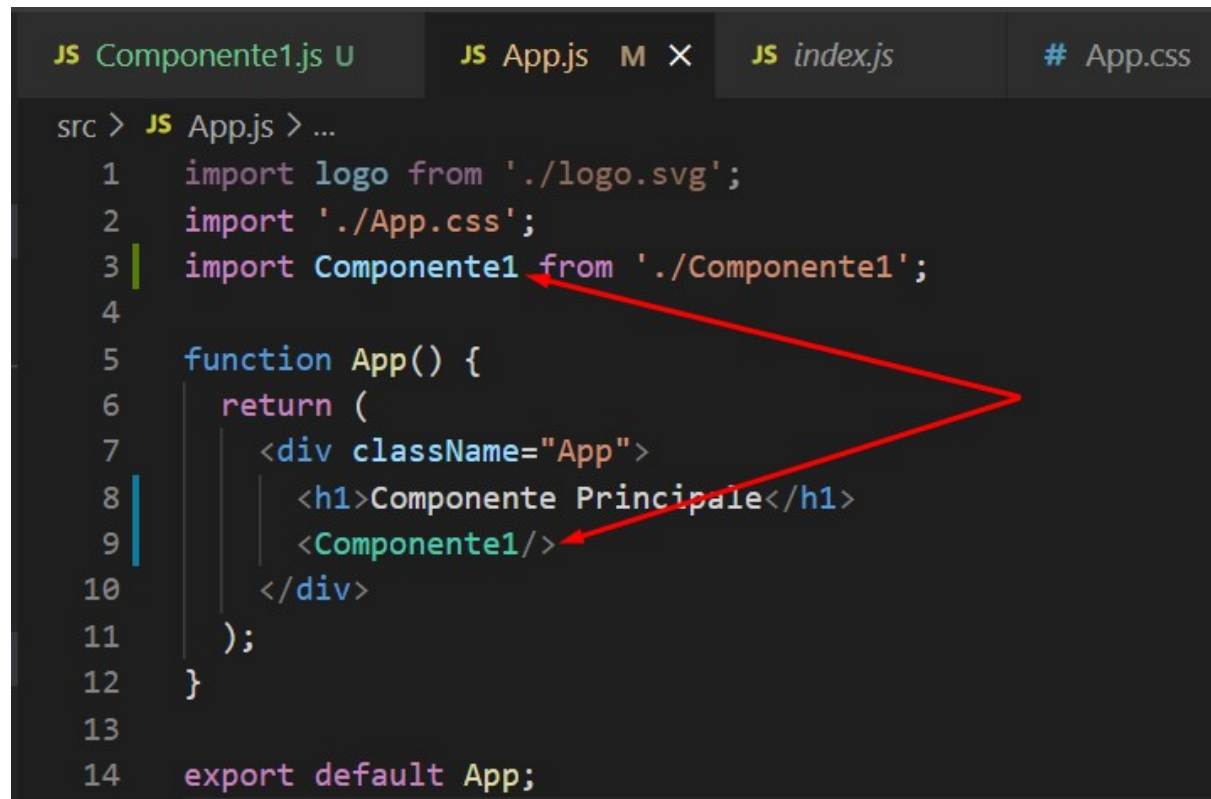
Il comando più usato del plugin appena installato è rafce (**R**ead **A**rrow **F**unction **E**xport **C**omponent) che ci facilita la scrittura dello scheletro di un componente :



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

```
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 (<>... </>)

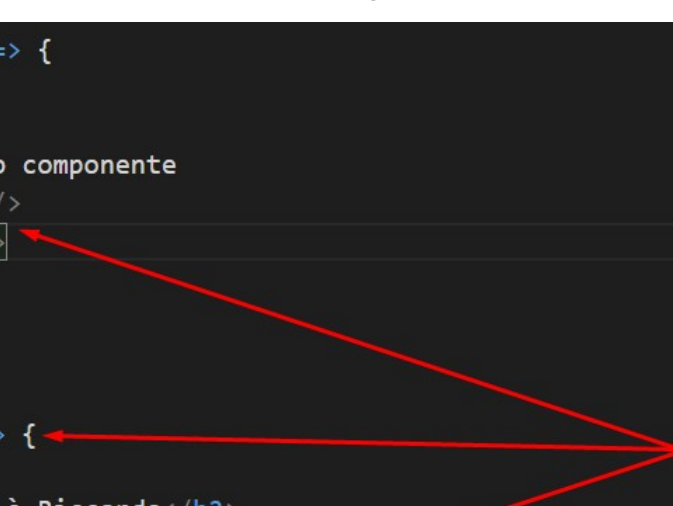
```
src > Js Componente1.js > [🔍] default
1  import React from "react";
2
3  const Componente1 = () => {
4    return (
5      <React.Fragment>
6        <h2>Ciao</h2>
7        <h3>Cio Ciao</h3>
8      </React.Fragment>
9    );
10 };
11
12 export default Componente1;
13
```

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 Innestanti

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 Inneitati

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';
```



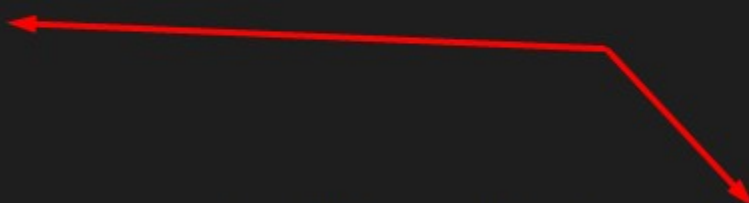
```
function App() {  
  return (  
    <div className="App">  
      <h1>Componente Principale</h1>  
      <Componente1/>  
      <Componente1/>  
      <Componente1/>  
      <Componente1/>  
    </div>  
  );  
}
```

```
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>  
  )  
}
```

A red arrow originates from the `{props.nome}` prop in the text "Il mio primo componente {props.nome} l'ho realizzato a {anni} anni" and points to the `<Anagrafica/>` component. Another red arrow originates from the `{anni}` prop in the same text and points to the `<Messaggio/>` component.

```
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>  
  );  
}
```

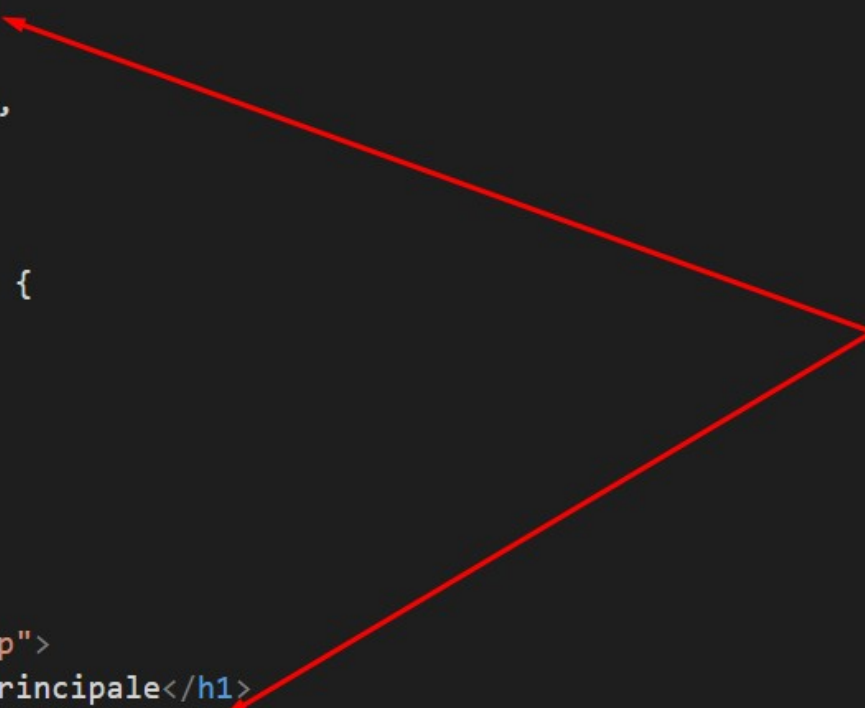
```
const Componente1 = (props) => {  
  
  return (  
    <div>  
      Il componente è di {props.nome} {props.nome} ed ho {props.anni} anni  
      <Anagrafica/>  
      <Messaggio/>  
    </div>  
  )  
}
```

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
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
```

# 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. x

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.

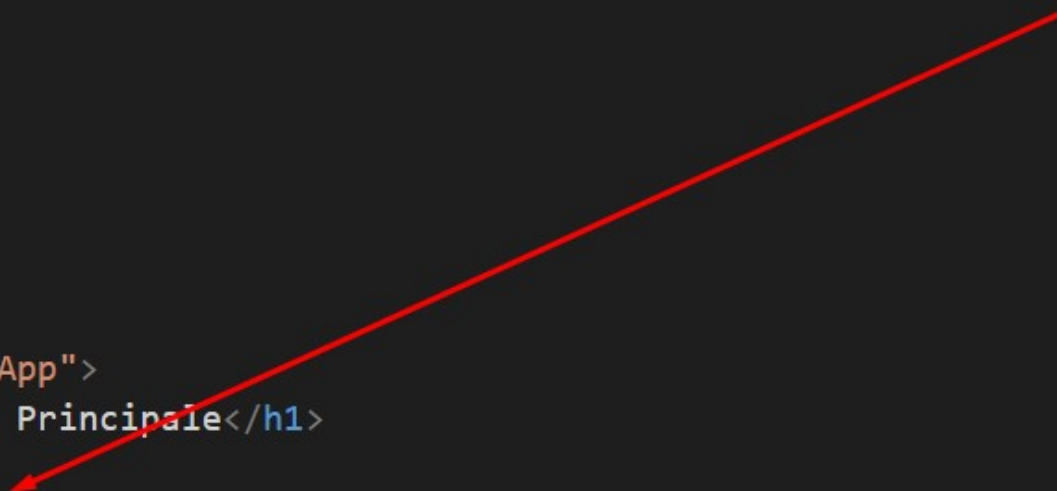
# map method

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>  
  );  
}
```



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>
  );
}
```

The diagram consists of three red arrows originating from a single point on the right side of the image. The first arrow points to the 'id : 1' property of the first object in the 'persone' array. The second arrow points to the 'id : 2' property of the second object in the 'persone' array. The third arrow points to the 'persone.map' call within the 'App' function, indicating that the 'persone' array is being passed as an argument to the 'map' method.

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 > [🔗] default

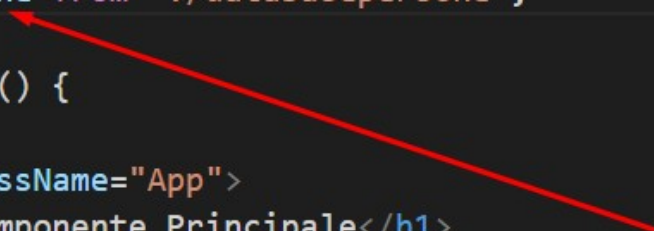
```
const persone = [{
  id : 1,
  nome : "riccardo",
  cognome : "cattaneo",
  eta : "30"
},{
  id : 2,
  nome : "mario",
  cognome : "rossi",
  eta : "35"
}];

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('inviato') }>INVIA</button>  
    </div>  
  );  
}  
  
export default App;
```



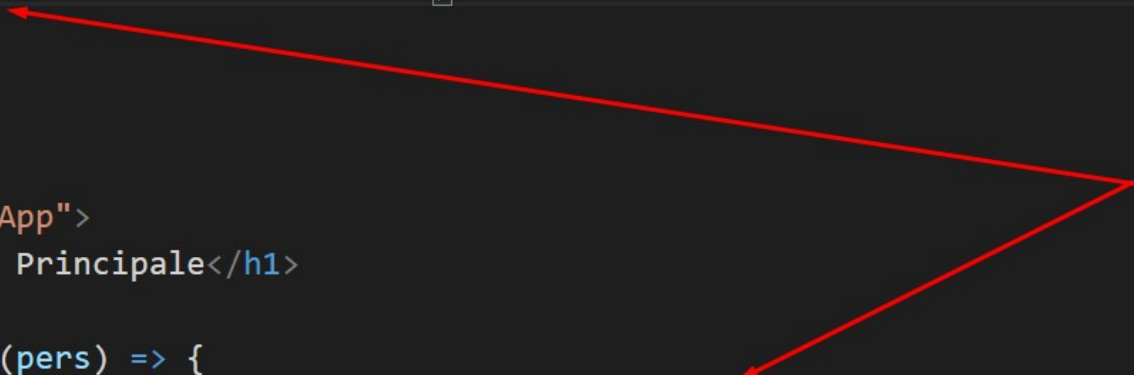
In questo primo esempio abbiamo scritto la nostra arrow function direttamente nel tag, ma possiamo anche creare una funzione con il codice e richiamarla al verificarsi dell'evento :

```
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>  
  );  
}
```

A red arrow originates from the 'invio' variable in the first code block and points to the 'invio' prop in the 'onClick' attribute of the button in the second code block. Another red arrow originates from the '=>' symbol in the arrow function definition and points to the '=>' symbol in the 'persone.map' function call.

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>  
  );  
}
```

A red arrow originates from the `invio(pers.nome)` call inside the `onClick` prop of the `<h1>` element in the JSX. It points diagonally upwards and to the left, ending at the `invio` function definition at the top of the code block.



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) }>  
        })  
      }  
      <button >INVIA</button>  
    </div>  
  );  
}
```

A red arrow originates from the `onMouseOver={ () => invio(pers.nome,pers.cognome) }` attribute in the JSX and points to the `const invio` function definition at the top of the code block.

# 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>  
    </>  
  )  
}
```

```
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