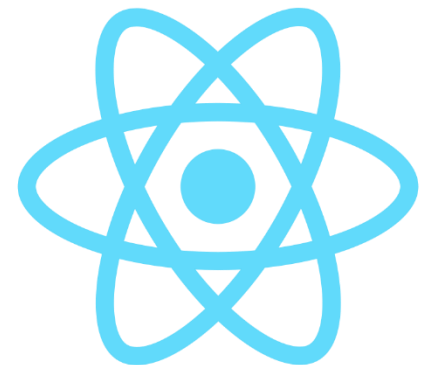


# React

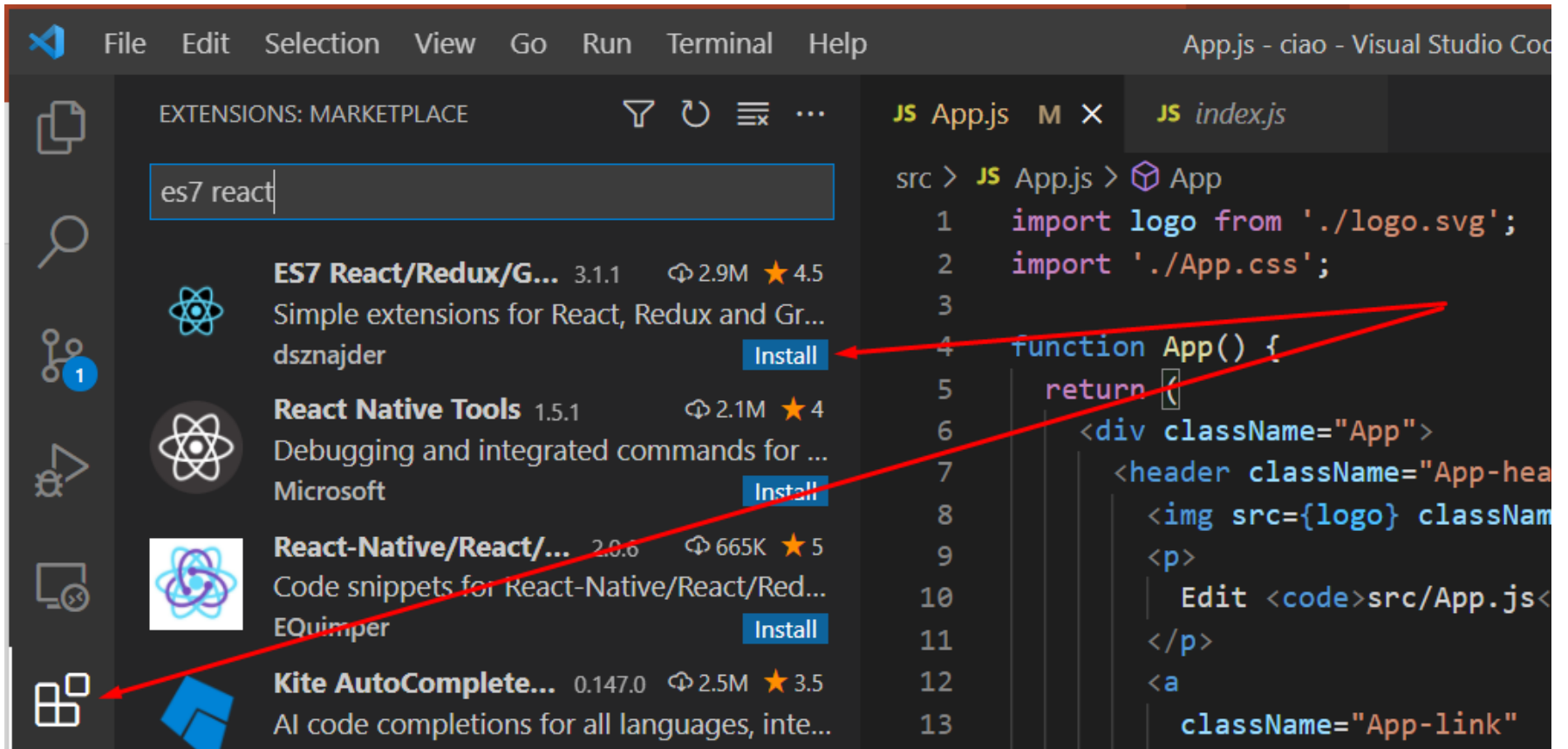
Riccardo Cattaneo



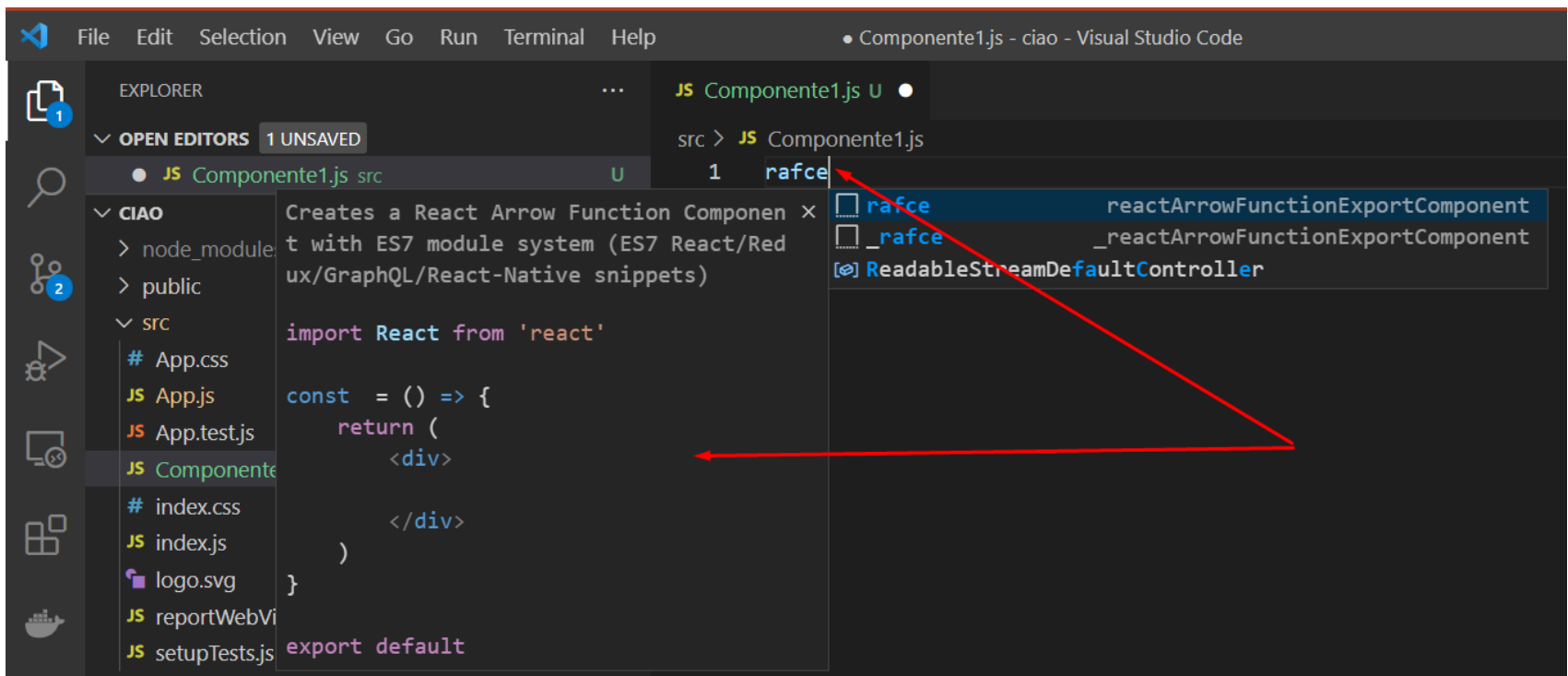
# Componente

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.



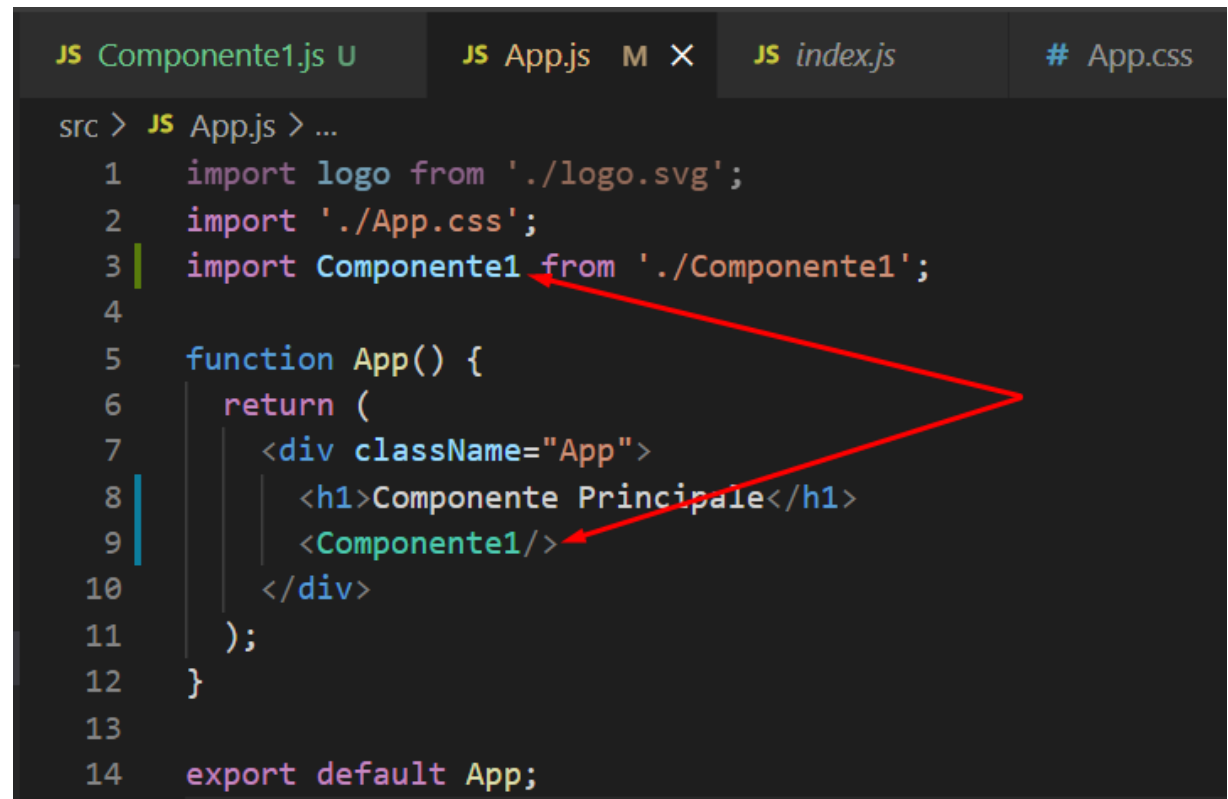
Il comando più usato del plugin appena installato è rafce (**R**ead **A**rrow **F**unction **E**xport **C**omponent) che ci facilita la scrittura dello scheletro di un componente :



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** .....

# 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>
```

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.

# 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
```

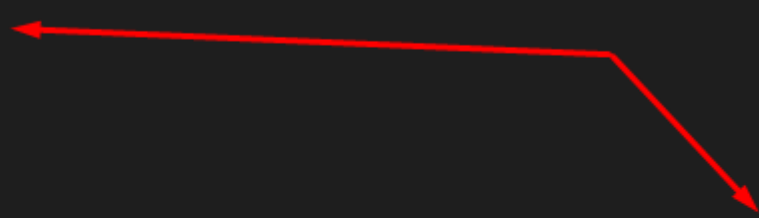
```
function App() {  
  return (  
    <div className="App">  
      <h1>Componente Principale</h1>  
      <Componente1/>  
      <Componente1/>  
      <Componente1/>  
      <Componente1/>  
    </div>  
  );  
}
```

```
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>  
  )  
}
```

A red arrow originates from the variable 'anni' in the line 'const anni = "30";' and points to the '{anni}' placeholder within the JSX element 'Il mio primo componente l'ho realizzato a {anni} anni', illustrating how the variable's value is passed into the component's output.

# 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 diagram with two red arrows. One arrow starts at the `{props.nome}` prop in the text 'Il mio primo componente {props.nome} l'ho realizzato a {anni} anni' and points to the `nome` prop in the `<Anagrafica/>` component. The second arrow starts at the `{anni}` prop in the same text and points to the `anni` prop in 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>  
  );  
}
```

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 :

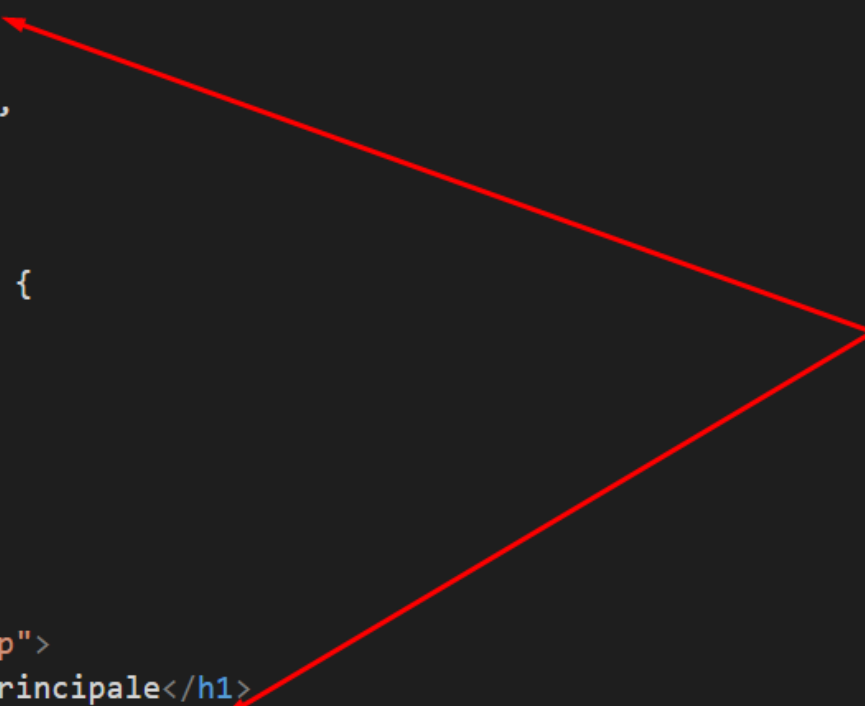
```
function App() {  
  return (  
    <div className="App">  
      <h1>Componente Principale</h1>  
      <Componente1 nome="riccardo" cognome="cattaneo" eta="30"/>  
      <Componente1 nome="mario" cognome="rossi" eta="35"/>  
      <Componente1 nome="lucia" cognome="verdi" eta="40"/>  
      <Componente1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



# Destrutturazione Oggetto

Vediamo ora come poter accedere in maniera più semplice alle proprietà di props attraverso la destrutturazione che avviene estrapolando ogni singolo elemento in questo modo

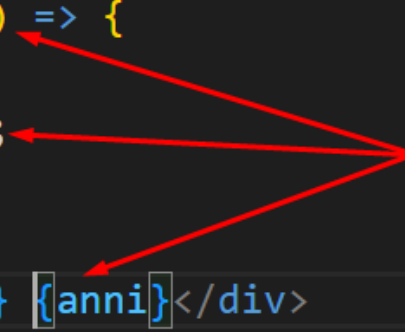
```
import React from 'react'

const Componente1 = (props) => {

  const {nome, anni} = props;

  return (
    <div>Componente1 {nome} {anni}</div>
  )
}

export default Componente1
```



The diagram consists of three red arrows originating from a single point on the right side of the code block. The first arrow points to the `props` parameter in the function signature `(props)` of the `Componente1` function. The second arrow points to the `props` variable in the destructuring assignment `const {nome, anni} = props;`. The third arrow points to the `{anni}` prop in the JSX element `<div>Componente1 {nome} {anni}</div>`.

Oppure possiamo destrutturare direttamente in ingresso alla funzione in questo modo :

```
import React from 'react'

const Componente1 = ( { nome, anni } ) => {

  //const {nome, anni} = props;

  return (
    <div>Componente1 {nome} {anni}</div>
  )
}

export default Componente1
```

A red arrow originates from a point on the right side of the code block and points to the 'anni' property in the destructured props object '{ nome, anni }' within the function parameter list. A second red arrow originates from the same point on the right and points to the '{anni}' prop in the JSX element '<div>Componente1 {nome} {anni}</div>'. This diagram illustrates how the 'anni' prop is passed from the function's parameters to the component's render output.

# 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 non visualizziamo nulla. Questo perché stiamo cercando di visualizzare un oggetto.



Questo avviene perché se vogliamo passare un'array, dobbiamo utilizzare la seguente sintassi

```
<Componente ut = {arrayutenti}/>
```

In secondo luogo, per poter visualizzare correttamente l'array a video devo utilizzare 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>
  );
}
```

A diagram with three red arrows originates from a single point on the right side of the image. One arrow points to the 'id : 1,' property in the first object of the 'persone' array. A second arrow points to the 'id : 2,' property in the second object of the 'persone' array. A third arrow points to the 'persone.map' call within the 'App' function's return statement.

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.



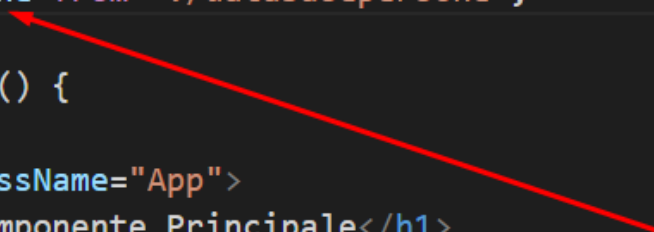
```
JS databasepersone.js > [🔗] default
const persone = [{
  id : 1,
  nome : "riccardo",
  cognome : "cattaneo",
  eta : "30"
},{
  id : 2,
  nome : "mario",
  cognome : "rossi",
  eta : "35"
}];

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;
```



# Esercizi

- 1 - Scrivere un componente che stampi la tabellina di un numero, compreso tra 1 e 10 a vostro piacimento.
- 2 - Scrivere un componente che stampa i numeri da 0 a 10 ;
- 3 - Modificare l'esercizio 2 in modo che stampa i numeri da 5 a 15 ;
- 4 - Scrivere un programma che conta da 0 a 20 con passo 2 e stampa i numeri ottenuti (0,2,...,20) ;
- 5 - Modificare l'esercizio 4 in modo che stampi il doppio dei numeri ottenuti (0,4,...,40) ;