Approfondimento: Closure Indipendenti con Esempi
==========

Quando eseguiamo piu volte la funzione `creaContatore()`, ogni chiamata crea una **nuova istanza indipendente** della variabile `contatore`. Questo succede grazie al concetto di **closure**.

Esempio:

```javascript
function creaContatore() {
  let contatore = 0;
  return function() {
    contatore++;
    console.log(contatore);
  };
}

const c1 = creaContatore();
const c2 = creaContatore();

c1(); // 1
c1(); // 2
c2(); // 1
```

Spiegazione passo-passo:
---

1. `const c1 = creaContatore();`
   - Chiama `creaContatore()`
   - Crea una nuova variabile `contatore = 0`
   - Ritorna una funzione che "ricorda" quella variabile `contatore`
   - Salva quella funzione in `c1`

2. `const c2 = creaContatore();`
   - Chiama DI NUOVO `creaContatore()`
   - Crea UNALTRA variabile `contatore = 0` (diversa da quella usata da `c1`)
   - Ritorna una nuova funzione con il proprio "ambiente chiuso"
   - Salva questa funzione in `c2`

Chiamate successive:

- `c1(); // 1` usa il `contatore` interno della sua closure (`contatore = 1`)
- `c1(); // 2` stesso `contatore`, ora vale 2
- `c2(); // 1` e una funzione diversa, con un altro `contatore`, che parte da 0 incrementato a 1

Perche `c1` e `c2` sono indipendenti?

Ogni chiamata a `creaContatore()`:

crea una nuova **funzione interna**
lega quella funzione al **suo ambiente chiuso**, contenente una **propria variabile `contatore`**

Quindi:
- `c1()` e `c2()` **sono due funzioni diverse**
- Ognuna **ha il proprio "mondo interno"**
- Le variabili usate da una **non influenzano** laltra

Visualizzazione concettuale:

Funzione	Ambiente chiuso	Valore interno
`c1`	`{ contatore: 0 }`	1 2
`c2`	`{ contatore: 0 }`	1

Ogni funzione mantiene un riferimento alla sua **variabile `contatore` privata**, anche dopo che `creaContatore()` e terminata.

Questo e il cuore delle **closure** in JavaScript:
la funzione restituita ha ancora accesso allo **scope in cui e nata**.